Grado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO FIN DE GRADO

Rediseño del Tutor Automático de un Laboratorio Virtual

Autor: Diego Dotor Jara

Tutor: Jaime Ramírez Rodríguez

MADRID, ENERO DE 2015

Índice

1. Resumen/Summary

El siguiente trabajo abarca todo el proceso llevado a cabo para el rediseño de un sistema automático de tutoría que se integra con laboratorios virtuales desarrollados para la realización de prácticas por parte de estudiantes dentro de entornos virtuales tridimensionales.

Los principales objetivos de este rediseño son la mejora del rendimiento del sistema automático de tutoría, haciéndolo más eficiente y por tanto permitiendo a un mayor número de estudiantes realizar una práctica al mismo tiempo. Además, este rediseño permitirá que el tutor se pueda integrar con otros motores gráficos con un coste relativamente bajo.

Se realiza en primer lugar una introducción a los principales conceptos manejados en este trabajo así como algunos aspectos relacionados con trabajos previos a este rediseño del tutor automático, concretamente la versión anterior del tutor ligada a la plataforma OpenSim. Acto seguido se detallarán qué requisitos funcionales cumplirá así como las ventajas que aportará este nuevo diseño. A continuación, se explicará el desarrollo del trabajo donde se podrá ver cómo se ha reestructurado el antiguo sistema de tutoría, la aplicación de un diseño orientado a objetos y los distintos paquetes y clases que lo conforman. Por último, se detallarán las conclusiones obtenidas durante el desarrollo del trabajo así como la implicación del trabajo aquí mostrado en futuros desarrollos.

The following work shows the process that has been carried out in order to redesign an automatic tutoring system that can be integrated into virtual laboratories developed for supporting students' practices in 3D virtual environments.

The main goals of this redesign are the improvement of automatic tutoring system performance, making it more efficient and therefore allowing more students to perform a practice at the same time. Furthermore, this redesign allows the tutor to be integrated with other graphic engines with a relative low cost.

Firstly, we begin with an introduction to the main concepts used in this work and some aspects concerning the related previous works to this automatic tutoring system redesign, such as the previous version of the tutoring system bound to OpenSim. Secondly, it will be detailed what functional requirements are met and what advantages this new tutoring system will provide. Next, it will be explained how this work has been developed, how the previous tutoring system has been restructured, how an object-oriented design is applied and the classes and packages derived from this design. Finally, it will be outlined the conclusions drawn in the development of this work as well as how this work will take part in future works.

2. Introducción

Los entornos virtuales, o mundos virtuales, desde su aparición han visto incrementado notablemente su impacto en el mundo de la simulación. Estos entornos, en los cuales se intentan representar escenarios reales mediante gráficos tridimensionales, son utilizados para una amplia gama de actividades. Dichas actividades pueden ir desde la recreación de edificios históricos, recreación de escenarios de crímenes, producción de películas de animación, videojuegos, simuladores (vuelo, conducción, etc.).

Los simuladores son un grupo de programas en los que se pretenden simular experiencias o actividades dentro de un entorno virtual. Estas simulaciones suponen, en muchos de los casos, una experiencia de aprendizaje muy valiosa al sumergir al usuario en un mundo que recrea, en mayor o menor medida, la realidad. Además, la realización de determinadas tareas en un entorno virtual conlleva un ahorro económico para la empresa o entidad que quiere entrenar en dichas tareas. Esto es debido a que en el mundo real el realizar dichas tareas puede acarrear considerables costos materiales, personales e incluso puede suponer un grave peligro para el usuario en caso de ocurrir un error durante la ejecución de dichas tareas.

Dados los problemas que presentaba la realización de ciertas prácticas de biotecnología en la realidad, en la Escuela Técnica Superior de Ingeniería de Montes, y gracias a la ayuda del Servicio de Innovación Educativa de la Universidad Politécnica de Madrid, se desarrolló con ayuda de miembros de la Escuela Técnica Superior de Ingeniería Informática un laboratorio virtual de biotecnología. Este laboratorio virtual permite al alumnado de la Escuela de Montes llevar a cabo una práctica dirigida a la adquisición de los conocimientos necesarios para la realización de esa misma actividad en un laboratorio de biotecnología real. Este laboratorio supondría no solo un ahorro de costes (el material necesario para la realización tienen un alto valor económico y los alumnos carecen de la habilidad para su correcta manipulación) para la Escuela sino también una ventaja para el alumnado al permitir que este, en su mayoría, pudiera acceder a esta práctica.

Dicho laboratorio fue desarrollado por miembros de la Escuela Técnica Superior de Ingeniería Informática de la Universidad Politécnica de Madrid. Para ello se hizo uso de la plataforma OpenSim, herramienta que permite la creación de entornos virtuales de manera sencilla. Se recrearon para ello toda la maquinaria y materiales necesarios para la realización de la práctica, programando además el comportamiento de estos mediante el lenguaje de *scripting* incluido en OpenSim, LSL. Para realizar una supervisión automatizada de las actividades que realizarían los alumnos dentro del laboratorio y comprobar así que éstos realizaban la práctica correctamente se diseñó e implementó un sistema automático de tutoría. Dicho sistema fue codificado en el mismo lenguaje que la funcionalidad de los objetos, además de hacer uso de una serie de ficheros complementarios.

Este sistema automático de tutoría o tutor automático, debido a que se encuentra integrado dentro del laboratorio de biotecnología y este en la plataforma de OpenSim, adolece de ciertas limitaciones que se decidieron subsanar con la realización de este trabajo fin de grado. Así, este trabajo consiste en rediseñar y reimplementar dicho tutor fuera de la plataforma de OpenSim utilizando para ello el lenguaje orientado a objetos C#. Extrayéndolo de OpenSim, y con el nuevo diseño realizado, se conseguirá una mejor mantenibilidad del código del tutor así como su portabilidad, ya que se podrá utilizar en otras plataformas de desarrollo de entornos virtuales, como podría ser Unity. Además, se ganará velocidad de ejecución de dicho sistema ya que OpenSim se encuentra desarrollado en C# y el tutor está implementado en el lenguaje LSL, que es propio de la plataforma y se ejecuta de forma interpretada, lo que supone un aumento de latencia. Otra ventaja será que podremos utilizar todos los *frameworks* que ya hay para C#, lo cual facilitará futuras extensiones. Una ventaja de este rediseño del tutor será el propio diseño orientado a objetos, que supondrá hacer al sistema más modular y más fácilmente extensible, pudiéndose modificar algunos de sus módulos sin que esto afecte en muchos casos al resto del sistema. Y para finalizar, se pretende integrar en el nuevo sistema una ontología con información sobre los estudiantes que será utilizada para almacenar los *logs* de las actividades de los alumnos durante la realización de la práctica. El uso de esta ontología ayudará, en un futuro, a realizar inferencias sobre estos *logs* permitiendo conocer el grado de conocimiento adquirido por el alumnado.

Por consiguiente, los objetivos de la realización de este trabajo serán:

- El rediseño e implementación en C# del sistema automático de tutoría haciéndolo externo a la plataforma OpenSim.
- Implementación de las extensiones necesarias para que el sistema automático de tutoría permita almacenar los *logs* de los estudiantes en la ontología del estudiante.

La estructura de esta memoria será la siguiente:

- Trabajos previos: En este apartado se explicará qué es un mundo virtual y cuáles son algunas de sus aplicaciones; qué es OpenSim y cuáles fueron sus orígenes; cómo es el laboratorio de biotecnología anteriormente mencionado así como el sistema automático de tutoría implementado en este y que será el punto de partida de este trabajo; qué es una ontología y cómo es el *framework* (Jena) que se utilizará para acceder a esta; y cómo es la ontología del estudiante y el uso que se hace de ella en este trabajo fin de grado.
- Especificaciones de requisitos: En este apartado se especificarán los requisitos exigidos al nuevo sistema automático de tutoría.

- Desarrollo: Aquí se detallará cómo se ha llevado a cabo el trabajo, es decir, el diseño del sistema automático de tutoría, la implementación de este, problemas surgidos durante el desarrollo del trabajo, etc.
- Conclusiones y futuro trabajo: En este apartado se expondrán las conclusiones a las que se ha llegado tras la realización del trabajo así como el impacto que tendrá dicho trabajo en futuros trabajos que se realizarán en nuestro laboratorio, además de sentar las bases para otros trabajos futuros.

3. Trabajos previos

3.1. Mundos Virtuales. Aplicaciones

Un mundo virtual [Wikimedia Foundation Inc. 2014a] es un entorno de simulación en el que se pretende simular un entorno más o menos real. Pueden estar desarrollados en entornos tanto de dos o tres dimensiones y los usuarios poseen una representación de sí mismos en estos mundos llamados avatares. Suelen ser mundos en los que se congregan un gran número de personas, cada uno representado por su avatar, y en donde pueden interactuar con el mundo, de una manera persistente, o con el resto de usuarios. Para ello han hecho uso de las redes de ordenadores para la interacción con otros usuarios. Además, estos mundos poseen una serie de reglas o funciones que integran, con mayor o menor acierto, las leyes físicas existentes en el mundo real.

Los mundos virtuales, tal y como se conocen ahora, nacen de la implementación de simuladores de la vida real. En 1973-1974 nació el que sería el primer juego que haría uso de estos mundos virtuales multi-jugador, el cual consistía en un grupo de jugadores cuyos avatares eran unos simples globos oculares que se desplazaban por un laberinto cazando y disparando al resto de jugadores. Este juego se jugaba sobre ARPANET, precursor de Internet. Los primeros mundos virtuales que hicieron uso de Internet fueron los conocidos como MUD (Multi-User Dungeon) [Wikimedia Foundation Inc. 2014b], los cuales consistían en mundos virtuales basados en texto en los que un grupo de usuarios, representados por avatares, llevaban a cabo juegos de rol y en los que podían comunicarse mediante *chat rooms* con el resto de jugadores para avanzar en la partida. Desde entonces los mundos virtuales, impulsados por las nuevas tecnologías y por la industria del videojuego, han crecido hasta cotas en las que se pueden simular mundos de una extensión geográfica considerable y en la que se pueden llegar a reunir hasta cientos, sino miles de jugadores al mismo tiempo. Algunos ejemplos de estos mundos virtuales serían:

- World of Warcraft, videojuego MMORPG (Massively Multiplayer Online Rol Playing Game) que cuenta con millones de suscriptores por todo el mundo, llegando a batir récords y que simula un mundo de fantasía de una importante extensión.
- The Sims, videojuego en el que se controla a una familia y el día a día de esta, así como la posibilidad de construir una casa al gusto del jugador.
- Second Life, mundo virtual en el que a los usuarios se les permite moverse por diferentes regiones, tener su propia región, construir en esta lo que se quiera y la programación de la funcionalidad de aquello que se construye.

- Grand Theft Auto, serie de videojuegos Sandbox en los que se permite al jugador/es recorrer un mundo virtual a su antojo, así como realizar aquellas actividades que quiera.
- Star Citizen, videojuego MMORPG en desarrollo que se está financiando a través de Kickstarter y que pretende simular un universo entero en el que el jugador podrá construir sus propias naves espaciales, su imperio, etc.
- Guild Wars y Lineage, otros videojuegos MMORPG.

Las aplicaciones de los mundos virtuales en la actualidad comprenden un gran abanico de campos. Algunas de estas aplicaciones pueden ser:

- En el sector del videojuego, como se puede apreciar en los ejemplos anteriores, siendo este sector uno de los que más apuesta por los mundos virtuales y de los que más han impulsado a estos.
- En el sector médico para tratar fobias. En estos casos se sumerge al usuario mediante dispositivos especiales como auriculares o cascos virtuales en una simulación en donde se les presentan sus mayores temores para, mediante numerosas sesiones, combatir esas fobias hasta el punto de dejar de padecerlas.
- En el sector del turismo virtual, donde se pueden representar míticos monumentos tal y como se construyeron en la antigüedad, realizando recorridos por sus recreaciones mientras un guía va contando hechos acaecidos en este escenario, cómo se construyó, etc.
- En el sector de la educación o el entrenamiento, mediante la simulación de prácticas que de otra forma no podrían ser realizadas por alumnos debido a su peligrosidad o costes.

3.2. Second Life /OpenSim

Second Life (SL) es un mundo virtual en el cual los jugadores controlan un avatar (representación virtual de una persona) y en el que con este avatar pueden recorrer los diferentes mundos o instancias que componen el mundo de Second Life. Fue desarrollado por Linden Lab [Linden Research, Inc., 2014b] inicialmente como software de apoyo para el que era su principal objetivo, la creación de hardware háptico, y que en un principio estaba más enfocado para el uso de armas, estilo *shooter*. Con el tiempo Linden Lab abandonó el desarrollo y la investigación de hardware háptico y se centró en el desarrollo de LindenWorld, que sería conocido a partir de entonces como Second Life. Se lanzó una beta privada en Noviembre del año 2002 y la beta pública en Abril del 2003. Ese mismo año fue lanzado oficialmente Second Life.

Figura 2.1. Second Life [Linden Research, Inc., 2014a].

Second Life consiste en una serie de regiones virtuales en las que los usuarios pueden construir todo aquello que quieran, también les da la posibilidad de visitar otras regiones, así como la total personalización de su avatar, todo ello limitado por los permisos que tenga el usuario dentro de Second Life. También posee un sistema económico interno, muy semejante al que podría darse en un país real. Este sistema económico se sustenta sobre una moneda virtual propia denominada Linden Dollar [Linden Research, Inc., 2014e] que se puede obtener a través de la plataforma LindenX [Linden Research, Inc., 2014f] pagando con divisas reales. Gracias a este sistema económico se han llegado a mover ingentes cantidades de dinero real dentro del mundo virtual, llegando algunos usuarios a ganar millones de dólares dentro de Second Life. Dentro del mundo virtual existe un mercado donde los usuarios pueden poner a la venta sus objetos, vestimentas, regiones y demás posesiones denominado Marketplace [Linden Research, Inc., 2014g].

Second Life posee su propio lenguaje de *scripting* con el cual dota de funcionalidad a los objetos que los usuarios pueden crear dentro del mundo virtual. Este lenguaje es denominado *Linden Scripting Language* (LSL) [Linden Research, Inc., 2014c] y posee una sintaxis similar al lenguaje C. LSL se basa en una serie de eventos que son programados por el usuario y que abarcan todas las posibles interacciones que pueden realizarse sobre el objeto en cuestión. Estas interacciones pueden ser desde tocar el objeto a la destrucción o creación del mismo.

OpenSim es una aplicación servidora multiplataforma y multiusuario en la cual se pueden desarrollar entornos virtuales. Fundado como proyecto en Enero del año 2007, el mismo año en el que se lanzó el cliente de Second

Life como *open source* así como libsl (librería para crear clientes personalizados). La idea original del proyecto era la de crear un servidor de prueba que pudiera conectarse con Second Life y que pudiera permitir realizar una serie de funciones básicas. Actualmente el objetivo es crear una plataforma para el desarrollo de entornos virtuales que pueda ser usada por varias aplicaciones como *framework*.

Figura 2.2. OpenSim [OpenSim Team, 2014a].

OpenSim otorga la oportunidad de visitar otros entornos virtuales de otros usuarios a través de una extensión denominada Hypergrid, con el único requisito de tener OpenSim instalado en la máquina local. OpenSim está escrito en el lenguaje C# permitiendo ser ejecutado sobre las plataformas Windows, MacOs y Unix. OpenSim usa el mismo protocolo de comunicación entre el cliente y el servidor que Second Life así como el mismo lenguaje de *scripting* LSL, por lo que mantiene cierta compatibilidad con este, permitiendo el uso del visor de Second Life para *renderizar* los entornos virtuales desarrollados en él. Debido a su diseño, OpenSim permite que los usuarios puedan incorporar extensiones en el servidor para facilitar el desarrollo de sus entornos. El sistema de tutoría que más adelante se describirá, será una extensión que se incorporará a OpenSim.

3.3. Laboratorio Biotecnología

Un laboratorio de biotecnología es un lugar especialmente habilitado para el desarrollo de experimentos biotecnológicos, es decir, posee maquinaria así como todo el material necesario (biológico y bioquímico) para la realización de experimentos dentro del campo de la biotecnología. Un laboratorio suele estar compuesto por diversas salas diseñadas para la realización de tareas concretas dentro del campo específico. Estas salas suelen ser:

- Sala de reuniones: estancia donde se organizan las pertinentes reuniones entre los miembros responsables del experimento o experimentos que se estén llevando a cabo dentro del laboratorio. Suele estar amueblado con una mesa grande alrededor de la cual se colocan las sillas donde tomarán asiento los miembros participantes de las reuniones. A su vez, también puede disponer de un proyecto así como de una pantalla para la visualización de diapositivas o vídeos que puedan tener alguna relación con los experimentos en desarrollo.
- Biblioteca: en esta sala se almacenan todos los libros que tengan información sobre la materia, así como puede también haber acopio de revistas y publicaciones relacionadas con el fin del laboratorio.
- Sala de seminarios: sala donde se imparten seminarios relacionados con la materia y que puedan ser de utilidad para los miembros componentes del equipo del laboratorio.
- Sala de computadores: en esta se encuentra el software necesario para la realización de cálculos o simulaciones, así como el permitir el acceso a publicaciones o información vía Internet que puedan ser de utilidad a los integrantes del laboratorio.
- Sala de crecimiento (fitotrón): lugar donde se almacenan los recipientes que albergan a las plantas para su crecimiento. Esta está ambientada con las condiciones óptimas (cantidad de luz, temperatura ambiente, humedad…) para el desarrollo de las plantas con las que se trabaja.
- Laboratorio o sala principal: sala en la que se desarrolla el/los experimento/s en cuestión y donde se lleva a cabo la manipulación de los materiales y en donde se concentra la mayor cantidad de dispositivos necesarios. Estos son clasificados según el nivel de seguridad requerido para llevar a cabo los experimentos, yendo desde el nivel P1 o el nivel de menor protección necesaria hasta el nivel P4 o el nivel de mayor protección necesaria.

En estas instalaciones, se llevan a cabo tanto experimentos como prácticas del alumnado. El problema surge cuando el alumnado carece de la experiencia y conocimiento para la manipulación tanto del material como de la maquinaria dando como resultado errores que no solo podrían poner en

peligro al alumno (en casos excepcionales de prácticas realizadas en laboratorios de nivel de seguridad P4) sino también acarrear costes económicos importantes, debido al elevado coste del material y de la maquinaria utilizados en el laboratorio, además del coste temporal que conlleva la realización de un experimento, pudiendo ser necesarios varios meses para su desarrollo. Por ello, se decidió implementar un laboratorio de biotecnología dentro de un entorno virtual tridimensional en el que el alumnado pudiera adquirir los conocimientos y nociones relacionados con la realización de las prácticas sin coste real. Dicho entorno debía simular con el mayor grado de detalle posible el desarrollo de la práctica así como el funcionamiento de la maquinaria y material necesarios. Para ello, se desarrolló dicho laboratorio [Riofrío, 2012; Roque, 2013; Fdez-Avilés, 2013] dentro de la plataforma OpenSim de forma que se pudiera realizar una práctica específica para el alumnado en la cual adquiriesen los siguientes conocimientos:

- Cultivo en condiciones in vitro, micropropagación de material vegetal y trabajo en condiciones de esterilidad, precauciones a tomar para evitar contaminaciones y utilización del material de laboratorio apropiado.
- Manejo de medidos de pH, balanzas de laboratorio y agitadores.
- Manejo de cabinas de flujo laminar.
- Utilización de autoclave.
- Preparación de medios de cultivo.
- Manejo de micropipetas de laboratorio.
- Fundamentos y manejo de la técnica de PCR.
- Fundamentos y técnicas para el clonaje de genes: ligación y clonación.
- Manejo de fitotrones.
- Técnica de transformación bacteriana mediante electroporación.
- Técnica de transformación vegetal.
- Técnicas de inoculación in vitro de especies vegetales.
- Análisis fenotípico de síntomas.

La práctica a realizar consistiría en la manipulación de un gen que codifica una proteína que funciona como antibiótico vegetal y que concedería a los individuos con los que se trabaja (chopos) resistencia a enfermedades, especialmente las de origen fúngico.

La práctica se divide en una serie de fases de manera que el alumno debe adquirir los conocimientos necesarios en las fases previas de cada fase. Estas fases contendrán una serie de acciones que el alumno deberá llevar a cabo para poder finalizar la fase en cuestión y que determinará el estado de desarrollo de la práctica, es decir, si se está llevando a cabo de la manera correcta o si, por el contrario, se han cometido errores. Estas fases y las acciones que las componen, están especificadas en el protocolo de la

práctica, previa creación de dicho protocolo por parte del profesor encargado de la supervisión de la práctica.

3.4. Tutor Automático

Para la supervisión de la realización de la práctica por parte del alumno se desarrolló e implementó un sistema automático de tutoría [Riofrío, 2012]. Dicho sistema, denominado Tutor Automático, se encarga tanto del seguimiento de la práctica del alumno como de guiar a este a lo largo de su realización. Para ello, el Tutor conoce el protocolo de la práctica así como una serie de mensajes asociados a los pasos de dicho protocolo que proporcionará al alumno en determinados casos. Se encargará pues de validar las acciones que realizase el alumno y, con arreglo al protocolo estipulado, notificar la existencia de algún error o no durante esa fase o bien enviar un mensaje que indique al alumno la siguiente acción a realizar. Toda la información manipulada por el Tutor está recogida en una serie de ficheros que contienen las acciones y fases que componen la práctica –con un determinado formato-, las acciones realizadas por los usuarios (*logs*), una lista de los objetos que pueden ser bloqueados (aquellos objetos que solo los puede utilizar un usuario hasta el final de la práctica o bien de manera temporal) y, por último, la fase en la que se encuentra cada estudiante. Por lo tanto, el Tutor registrará la actividad de cada estudiante y la cotejará con el protocolo de la práctica para realizar la acción de tutoría correspondiente.

Figura 2.3. Sala principal del laboratorio de biotecnología [Riofrío, 2012].

3.5. Ontología

Una ontología es una jerarquía de términos usados para describir un dominio que puede ser usado para construir una base de conocimiento, es decir, una base en la que se almacenan datos y la información que un ser humano obtendría de estos. Las ontologías son una manera de manejar información y datos de tal manera que tanto un humano como una máquina pueda interpretarlos. Para ellos, en el ámbito de la informática, una ontología define:

- Un vocabulario común de términos.
- Alguna especificación del significado de los términos.
- Una comprensión compartida por personas y ordenadores.

Una ontología posee una serie de cualidades que permiten:

- La reutilización de conocimientos.
- Mejorar la interoperabilidad.
- Transformar conocimiento implícito en explícito.
- Separar el conocimiento declarativo del procedimental.
- Conceptualizar y analizar el dominio de conocimiento.

Estas cualidades están definidas por los componentes que posee una ontología. Dichos componentes se encargan de mantener la información y la coherencia de estos en el dominio de conocimiento que se desea plasmar a través del uso de las ontologías. Los componentes que conforman una ontología son los siguientes:

- Clases: son las entidades primarias del mundo. Definen objetos, seres, eventos, tareas, etc.; representan un conjunto de elementos con propiedades similares (por ejemplo, la clase mamífero); y suelen constituir una jerarquía en la que unas clases son herederas de otras, es decir, comparten las mismas propiedades que la clase padre, pero poseen unas propiedades únicas que la definen como una clase en sí, usualmente definidas por la relación es-un (por ejemplo, ser humano es un mamífero).
- Atributos: son las propiedades inherentes de una clase y que la caracterizan (por ejemplo, altura, color, forma, raza, sexo…). Estos atributos pueden ser simples, es decir, tienen valores de primitivas (cadenas de caracteres, números, etc…) o bien complejos, en cuyo caso apuntan a otra clase. Además, pueden ser monovaluados (poseen única y exclusivamente otra instancia como valor) o multivaluados (pueden poseer una o más instancias como valor).

- Relaciones: son la forma en como dos clases se encuentran relacionadas (por ejemplo, un padre tiene hijo, donde "tiene" sería la relación entre la clase padre y la clase hijo).
- Axiomas: proposiciones más complejas desde el punto de vista lógico (por ejemplo, la transitividad en ciertas relaciones: tenemos p1, p2 y p3, si p1 es parte de p2 y p2 es parte de p3, entonces p1 es parte de p3).
- Instancias: representan elementos de las clases.

Las ontologías suelen utilizar los lenguajes basados en lenguajes de marcado para la creación de los ficheros que contengan toda la información relacionada con la ontología. Algunos lenguajes utilizados son XML, RDF (Resource Description Framework) y OWL (Ontology Web Language). Este último lenguaje ha sido adoptado en la actualidad como lenguaje principal para el desarrollo de ontologías. Para la creación de las ontologías se suelen utilizar una serie de editores que abstraen el sistema de etiquetado de los lenguajes, facilitando al usuario la implementación de la ontología gracias a una representación sencilla de las clases, las propiedades y la creación de instancias, además de otorgar una visualización de la jerarquía de clases. Algunos de estos editores son Protégé (Musen, Stanford), quizás el editor más utilizado para la creación y modificación de ontologías, y TopBraid[1].

3.6. Jena

Apache Jena es un *framework* de Java diseñado para la construcción de aplicaciones de Web Semántica, proveyendo un gran número de librerías para el manejo de ficheros RDF, RDFS, RDFa, OWL y SPARQL. Incluye también varios motores de inferencia que permiten implementar sistemas de razonamiento sobre ontologías. Jena es un proyecto de código libre que fue desarrollado originalmente por HP Labs en Bristol, Reino Unido, en el año 2000. En el año 2009, HP decidió reenfocar el desarrollo del proyecto para darle un desarrollo directo a este. En Noviembre del año 2010, Jena fue adoptado por Apache Software Foundation.

Jena trabaja sobre tripletas RDF que carga desde los ficheros con las extensiones anteriormente citadas y que mantiene en memoria para agilizar las consultas así como la creación de nuevas instancias, clases, reglas, relaciones, etc. Además, Jena posee un sistema de consulta SPARQL que permite consultas con una sintaxis semejante a la que se podría utilizar en una base de datos con SQL.

[1] http://www.topquadrant.com/tools/IDE-topbraid-composer-maestro-edition/

3.7. Ontología del Estudiante

La ontología del Estudiante [Clemente et al., 2011] es una ontología diseñada por un miembro del Laboratorio Decoroso Crespo en su tesis doctoral. La finalidad de dicha ontología era la de poder ser usada para el seguimiento de la adquisición de conocimientos de estudiantes en la ejecución de prácticas o a lo largo de la trayectoria de estudios del estudiante.

Esta ontología está diseñada modularmente, es decir, se compone de un grupo de ontologías más pequeñas disociadas parcialmente entre sí que permite los cambios en alguno de sus módulos sin que los cambios afectasen a la totalidad de la ontología del Estudiante. Dichos módulos son:

- *Student Information Ontology*: es la ontología raíz, en la que se contendría toda la información de un estudiante. Estaría compuesta por elementos de cada uno de los módulos restantes.
- *Student Monitoring Ontology*: es la ontología que se encarga de definir clases relativas a como se va a monitorizar el comportamiento del estudiante.
- *Learning Objective Ontology*: Es la ontología donde se define la clase objetivo de aprendizaje. Se distinguen dos subclases de objetivo de aprendizaje: *Didactic_Objective*, que abarca de forma general los objetivos de aprendizaje, tales como los obtenidos en una fase de un curso. Y *Specific_Objective*, que abarca los conocimientos relacionados con conocimientos específicos obtenidos de ejercicios o de actividades.
- *Knowledge Object Ontology*: en ella se definen las clases que van a permitir representar dentro de la ontología el conocimiento en sí del estudiante. Estos conocimientos pueden ser de tipo estructural como un concepto, una proposición, una relación, etc. Además, también puede definir conocimientos de tipo procedimental, como pueden ser acciones, planes, recorridos, etc. Las acciones vienen definidas por una completísima taxonomía que clasifica los diferentes tipos de acciones según el contexto.
- *Student Trace Ontology*: esta ontología permite representar la actividad del usuario a lo largo del curso. La clase más concreta de la jerarquía sería *Action_Trace* que sería la que almacenaría la información obtenida al ejecutarse una *Punctual_Action* (clase definida en *Knowledge Object Ontology*).
- *Student State Ontology*: define clases relacionadas con el estado del estudiante. Algunos de estas clases tienen que ver con el estado de conocimientos del estudiante, estado emocional del estudiante, etc.

- *Student Profile Ontology:* contiene la información demográfica del estudiante tales como el nombre de este, apellidos, dirección, código postal, etc.

La ontología del estudiante forma parte de un modelo, denominado modelo del estudiante, que además incluye un conjunto de reglas que permiten inferir el grado de adquisición de conocimientos logrado por un alumno a partir de su comportamiento en un entorno virtual de aprendizaje.

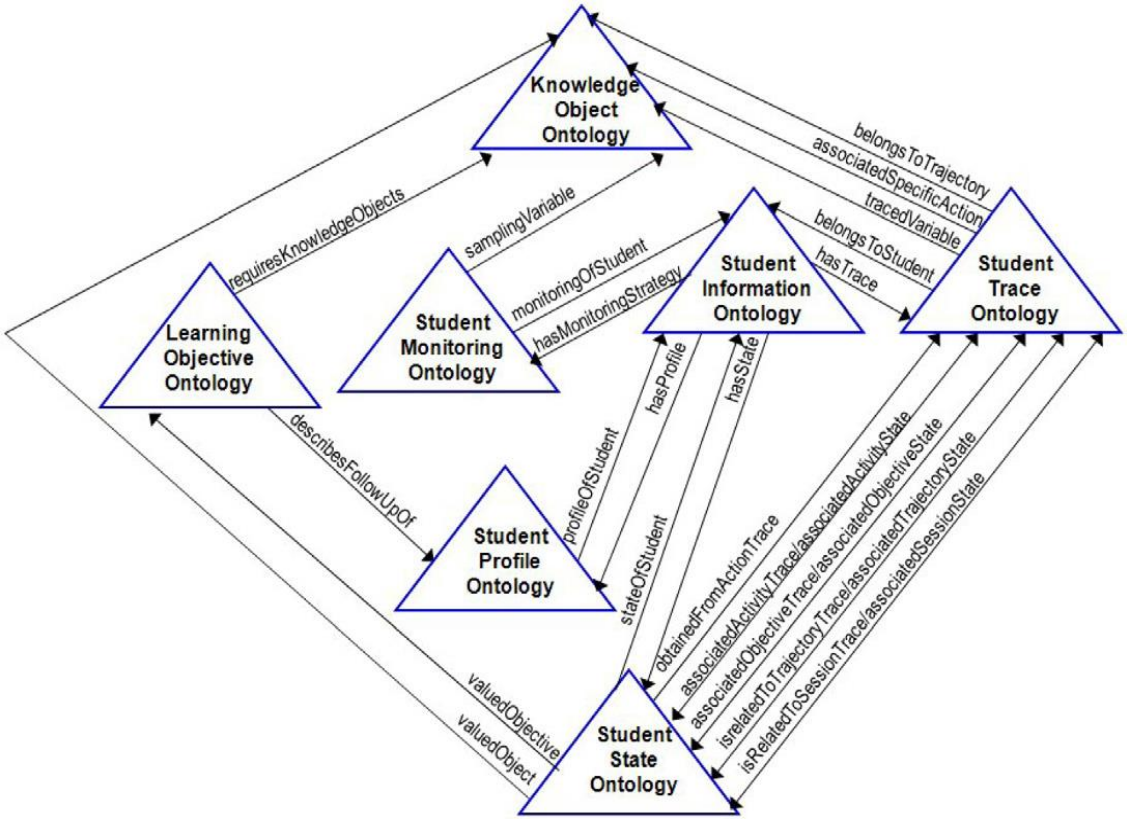

Figura 2.4. Descomposición modular de la ontología del estudiante [Clemente et al., 2011].

4. Especificación de Requisitos

En este apartado se procederá a detallar los requisitos que el sistema de tutoría deberá de cumplir. Estos requisitos se dividirán en dos secciones: especificación funcional y especificación no funcional.

4.1. Especificación Funcional

4.1.1. Funcionamiento deseado

El sistema de tutoría que se ha procedido a reconstruir y que se encuentra implementado dentro del entorno virtual de OpenSim posee el funcionamiento que se va a explicar a continuación. Dicho funcionamiento de cara al estudiante y a los objetos del entorno virtual que interaccionan con el tutor no se verá afectado por la reconstrucción. De esta manera, se podrá sustituir el anterior tutor por el nuevo sin que este cambio afecte al resto del laboratorio virtual (véase el requisito no funcional de Integración con OpenSim).

El sistema de tutoría o tutor [Riofrío, 2012], se compone de un objeto primitiva dentro del entorno que posee un *script* y una serie de ficheros externos de apoyo –que se explicarán más adelante. El script posee el código que maneja dichos ficheros y además es el encargado de recibir y responder las peticiones asociadas a las acciones de los avatares de los alumnos. Estas peticiones son las validaciones de las acciones llevadas a cabo por los alumnos y que el tutor deberá comprobar con respecto al protocolo de la práctica. Las peticiones son enviadas al tutor a través de un canal de mensajes interno del entorno virtual y que se corresponde con el canal 1 y cuyos emisores son los objetos del laboratorio que son parte de la práctica, es decir, aquellos objetos con los que interactúa el avatar. Estas peticiones pueden ser de uno de los siguientes tipos: *crearLibro*, *borrarFase*, *validar* o *borrar*. Dependiendo del tipo de petición, el tutor realizará una serie de acciones:

- *crearLibro*: se recibirá cuando el alumno inicie la práctica, y en este caso, el tutor creará una *notecard*, a la que llamaremos nota del cuaderno de protocolo, para el avatar que haya solicitado la petición. En dicha nota, se almacenarán las acciones que realizará este avatar durante la práctica.
- *borrarFase*: esta petición borrará por completo la última fase realizada por el avatar, es decir, se eliminarán todas las acciones registradas en la nota del cuaderno de protocolo que pertenezcan a esa última fase (eliminando la nota si tan solo contuviera la fase inicial).

- *validar*: en caso de recibir esta petición, el tutor llevará a cabo una serie de comprobaciones para dar o no por validada la acción enviada. Dichas comprobaciones serán las siguientes:

    1. Inicialmente se comprobará que el objeto requerido por el avatar no está siendo utilizado por otro avatar. En el caso de que el objeto no esté disponible, se mandará la respuesta a través del canal 2 al objeto que solicitó la validación de la acción y se notificará al usuario dicho bloqueo mediante un mensaje de error.
    2. En caso de que el objeto no estuviera bloqueado por otro usuario, el tutor comprobará si la acción a validar se ejecutó previamente. De haberse ejecutado y no estar permitida su ejecución más de una vez, al igual que en el paso anterior, se notificará al avatar mediante un mensaje de error.
    3. Tras esto, el tutor comprobará una por una cada una de las dependencias (acciones que deben haberse ejecutado previamente para que la acción en ejecución se realice correctamente) que tiene la acción, de tenerlas, y se verificará que estas hayan sido realizadas. En caso negativo, se tendrá en cuenta si la dependencia no realizada es bloqueante o no:
        a. Si es bloqueante, se procede a notificarlo mediante un mensaje de error y se bloquea el avance del usuario hasta realizar la acción de la que depende la actual,
        b. En caso contrario, se guarda en la lista de errores de dependencia.
    4. A continuación, el tutor comprobará la existencia de incompatibilidades, es decir, que se hayan ejecutado previamente acciones con las que la acción a validar sea incompatible. En el caso de que existan, se comprobará si son o no bloqueantes: en caso de que sean bloqueantes, se notifica con un mensaje de error, como se ha explicado anteriormente, al usuario y se anota el identificador junto al objeto con el que se ha interactuado en la *notecard* de bloqueos, y si no fuera bloqueante, se anotará el error en la lista de errores de incompatibilidad.
    5. También se comprobará si la acción debe verificar los errores no bloqueantes cometidos previamente, es decir, si la acción deberá comprobar la existencia de errores cometidos en acciones anteriores pero que el sistema de tutoría permitió al usuario continuar sin bloquear el progreso de este, mostrando todos los errores cometidos hasta entonces.

  Una vez que el tutor haya dado por válida la acción, se comprobará además si la acción implica un cambio de fase, cambiando la fase en

la que se encuentra el usuario en caso afirmativo. Además, en caso de que la acción tenga asociado un temporizador, ya sea de mínimo o de máximo tiempo, el tutor se encargará de gestionar los tiempos para dichas acciones. Estos temporizadores indican un tiempo, en segundos, que deberán pasar en el caso de que sea mínimo, o que como mucho pueden pasar en caso de que sea máximo, para dar a la acción como correcta. Para ello, el tutor anotará en la nota correspondiente un sufijo que identifique la acción. Por último, el tutor guardará en la nota del cuaderno de protocolo del usuario la acción realizada así como visualizará el mensaje de pista para la siguiente acción, en caso de existir dicho mensaje en la especificación de la tutoría.

- *borrar*: el tutor borrará la acción de la nota del cuaderno de protocolo del usuario, debido a un error reportado por el objeto.

Tras explicar el funcionamiento del tutor que se pretende mantener, se procede a detallar las entradas y salidas que poseía el tutor, para acto seguido explicar cuáles serán las nuevas entradas y salidas del tutor una vez se reconstruya.

4.1.2. Especificación de las entradas y salidas

El tutor posee una serie de entradas y salidas que son parte del tutor o apoyan el funcionamiento de este. Estas entradas y salidas pueden ser mensajes o escrituras/lecturas de ficheros, y se detallan a continuación:

- **Nota de la Estrategia de Tutoría** (entrada). Se trata de un fichero interno del entorno virtual de OpenSim que se denomina *notecard* y cuyo funcionamiento es básicamente igual al de un fichero de texto plano. En este fichero se encuentran listadas las acciones que se deben de realizar para llevar a cabo la práctica en cuestión. Cada línea del fichero es una entrada o acción que posee una serie de campos que la definen y que se encuentran separados por el carácter barra vertical (|) para su posterior tratamiento. Dichos campos son:

    - Código secuencial de la acción. Código único que identifica la acción y que está compuesto por la fase a la que pertenece la acción así como el número de la acción.
    - Código descriptivo. Es el código por el cual las acciones son buscadas por el tutor tras recibirse un mensaje procedente de los objetos para su validación.
    - Mensaje de aprobación. Este campo posee su propia estructura interna y posee el siguiente formato: *número (1 o 0)* –

mensaje. El número indica si el mensaje es o no mostrado al avatar tras validar correctamente la acción (1 y 0 respectivamente), mientras que el mensaje es el texto que se debe mostrar al usuario.

- Mensaje de la siguiente acción. Es un texto que se puede mostrar o no como ayuda al usuario para indicar a este cuál es la siguiente acción que deberá realizar.
- Códigos de las acciones de dependencia. Contiene los códigos secuenciales de las acciones con las que tiene un tipo de dependencia. Estas dependencias se separan con guiones en caso de existir más de una, y a su vez, se las acota dentro de unos corchetes en caso de que el orden de su realización no sea relevante, y entre paréntesis cuando el orden de su realización sí importa y debe coincidir con el especificado.
- Mensaje de error por falta de alguna dependencia. Contiene tantos mensajes como dependencias existan en el apartado anterior teniendo cada mensaje la misma estructura que el mensaje de aprobación. En este caso, los números indican si se bloquea el progreso o no en la práctica. Cada mensaje, con su estructura correspondiente, es separada del siguiente mediante el carácter barra inclinada (/).
- Mensaje de error por orden de las dependencias. Este mensaje posee el mismo formato que el explicado en el punto anterior, y mantiene la misma característica bloqueante. Se trata de un mensaje que se debe mostrar al usuario en caso de no haberse realizado las dependencias en el orden especificado en el apartado "códigos de las acciones de dependencia".
- Acciones incompatibles. Contiene los códigos secuenciales de las acciones que son incompatibles con la acción actual. Estos códigos se separan mediante guiones.
- Mensaje de error de incompatibilidades. Al igual que los mensajes de error de dependencias, son los mensajes que se deben mostrar al usuario en caso de haberse llevado a cabo una de las acciones incompatibles previamente. La estructura de este apartado es el mismo que el de "mensajes de error por falta de alguna dependencia", al igual que el cometido de cada parte de esta.
- *Flag* de bloqueo o desbloqueo. Tiene como valores el 1 y el 0 que indican si el objeto que envió la petición de validación es bloqueado o desbloqueado, respectivamente, por la acción. En caso de que la acción ni bloquee ni desbloquee, el campo permanece en blanco. Los bloqueos y desbloqueos se anotan en la nota de bloqueos, detallada más adelante.

  - Tiempo máximo. Determina el tiempo máximo, en segundos, que puede transcurrir desde que se realice esta acción hasta la ejecución de otra acción que la tiene como dependencia.
  - Tiempo mínimo. Determina el tiempo mínimo, en segundos, que debe esperarse hasta poder ejecutar otra acción que la tiene como dependencia.
  - *Flag* para validar repetición de la acción. Indica si una acción puede ser repetida (0) o si por el contrario solo se permite su ejecución una única vez (1).
  - *Flag* para validar errores cometidos. Indica si tras realizarse la acción se debe comunicar al usuario (1) los errores cometidos anteriormente y que no habían sido comunicados.
  - *Flag* que indica el final de una fase. Indica (1) si la acción es la última de una fase.

- **Nota de Bloqueos** (entrada/salida). Se trata de un fichero interno del entorno virtual (*notecard*) al igual que el anterior. Este contiene una línea con cada objeto que puede ser bloqueado por algún usuario seguido de una barra vertical (|), y después el identificador que lo bloquea o de nada en caso de que esté disponible. De este fichero se leen las líneas cada vez que se pretende validar una acción por parte del tutor; y se escribe en él siempre que el tutor valida una acción que bloquea un objeto.

- **Nota del Cuaderno de Protocolo** (salida). En este fichero (*notecard*) se guardan las acciones validadas por el tutor y realizadas por un determinado usuario. Cada línea del fichero corresponde a una de estas acciones y contiene los siguientes campos: Código secuencial de la acción, código de la acción, tiempo en el que se realizó la acción (para aquellas acciones que tenían un tiempo), código secuencial de las dependencias que permiten seguir con la práctica que no se han ejecutado y código secuencial de las acciones incompatibles que se han realizado, pero que permiten seguir con la práctica.

- **Nota de la Fase Actual** (salida). En dicho fichero (*notecard*) se almacenan los identificadores de los avatares de los usuarios y la fase de la práctica en la que se encuentra cada uno. A cada uno le corresponde una línea del fichero.

- **Mensaje de validación** (entrada). Los mensajes recibidos desde los objetos del entorno virtual que son enviados cuando el usuario interacciona con estos. Estos mensajes solicitan al tutor la validación

de la correspondiente acción realizada por el usuario y que tendrá, por parte del tutor, una respuesta.

- **Mensaje de respuesta** (salida). Se trata del mensaje de respuesta del tutor una vez que este ha validado una acción recibida mediante un mensaje de validación. Dicha respuesta puede tratarse de un mensaje de aceptación, es decir, la acción ha sido validada sin ningún problema, y un mensaje de tutoría, siendo este mensaje una pista para el usuario del cuál es la siguiente acción a realizar; o bien un mensaje de error debido a que el objeto está bloqueado, faltan por realizar una o más acciones de las que depende, existe algún tipo de incompatibilidad, o bien el tiempo mínimo o máximo asociado con la acción, de existir, se ha cumplido y por lo tanto, la acción pasa a ser una acción con errores de tiempo.

Teniendo en cuenta las entradas anteriores, que son las antiguas entradas y salidas del sistema de tutoría, se realizaron cambios en estas entradas y salidas para la nueva reconstrucción del tutor. Estas nuevas entradas y salidas poseen una cierta similitud, en cuanto a funcionamiento se trata, con las anteriores entradas y salidas. Las nuevas entradas y salidas serán:

- **Fichero de configuración de la Tutoría** (entrada): Se trata del homólogo a la Nota de la Estrategia de Tutoría. Debido a que el nuevo sistema de tutoría no debe acoplarse con ningún motor gráfico en concreto, se necesita externalizar los ficheros o notas anteriores. Este fichero se trata de un fichero Excel de extensión .xlsx y que posee, al igual que la Nota de la Estrategia de Tutoría, la lista de acciones que conforman una práctica. Cada acción se encuentra definida en una fila del fichero y cada fila posee una serie de campos en cada columna:

    - La primera columna corresponde al número de la fase en la que se encuentra la acción.
    - La segunda columna es una columna inútil en cuanto a la configuración del sistema de tutoría, ya que solo funciona internamente dentro del fichero para la construcción de los códigos de las acciones.
    - La tercera columna se trata del código identificador de la acción. Este código identificador es único y se construye utilizando la fase en la que se encuentra la acción y el número de esta.
    - La cuarta columna se corresponde con el código de la acción, que se trata como nombre de la acción.

- La quinta columna se corresponde con el mensaje de aceptación, y posee una estructura similar a los mensajes de la Nota de Estrategia de Tutoría. Este mensaje es el que devolvería el tutor cuando se valida una acción correcta.
- La sexta columna es la correspondiente a las posibles acciones siguientes. Se compone de una lista de los códigos identificadores de las acciones que pueden venir a continuación de la acción actual.
- La séptima columna contiene el mensaje de tutoría, es decir, el mensaje que se muestra cuando se ejecuta la acción.
- La octava, la novena y la décima columna se corresponden con las dependencias. La octava contendría la estructura con las dependencias de la acción, con el mismo formato que en la anterior versión. La novena y décima son los mensajes individuales de las dependencias y los mensajes de orden de las dependencias respectivamente. El formato de los mensajes es el mismo que en la versión anterior en todos los mensajes con la salvedad de que en este nuevo fichero la separación se realiza con el carácter de barra invertida (\).
- La undécima y duodécima columnas contienen las incompatibilidades y los mensajes asociados a dichas incompatibilidades. Las incompatibilidades mantienen la misma estructura de los códigos identificadores de las acciones separados por guiones. Los mensajes conservan el mismo formato que para los mensajes de dependencias.
- La decimotercera columna contiene el *flag* que indica si la acción bloquea o desbloquea el objeto asociado. Si bloquea el *flag* será un 1 y si desbloquea será un 0. No contendrá ni uno ni otro número si no bloquea/desbloquea la acción.
- La decimocuarta y decimoquinta columna son las que contendrán el tiempo máximo y mínimo, respectivamente, asociados a la acción.
- La decimosexta y decimoséptima columna son los mensajes de error asociados a los tiempos de las dos columnas previas. Estos mensajes conservan el formato de todos los mensajes vistos hasta ahora.
- La decimoctava columna es el *flag* que indica si la acción se debe ejecutar una sola vez o si se puede repetir indefinidamente.
- La decimonovena columna es el *flag* que indica si la acción actual es la que se encarga de validar los errores cometidos en la fase actual hasta dicha acción.

  - La vigésima columna es el *flag* que indica el cambio de fase, es decir, si la acción actual es la primera acción de una nueva fase.
  - La vigésimo primera columna y última columna, contiene el nombre del/de los objeto/s asociados a esta acción, es decir, el/los objeto/s que se deben utilizar para la realización de dicha acción.

- **Ontología del Estudiante** (entrada/salida). El conjunto de ficheros con extensión .owl que conforman la ontología del Estudiante serán parte de las nuevas entradas y salidas del tutor. Como salida, la ontología del Estudiante hará las funciones de almacén de los *logs*, es decir, en ella se almacenarán las trazas de los estudiantes con cada acción realizada, los errores cometidos, etc. El uso de esta ontología como almacén de las trazas es debido a que el sistema de tutoría se encontrará integrado en un futuro en un sistema de tutoría más grande y más complejo que utilizará al tutor diseñado en este trabajo. Dicho tutor más complejo hará uso de las trazas almacenadas en dicha ontología y los mecanismos inherentes de las ontologías para llevar a cabo inferencias sobre estas trazas, utilizando los resultados de estas inferencias para proporcionar una tutoría más compleja y personalizada. Asimismo, la ontología servirá para la recuperación de las trazas de los usuarios en el caso de que el sistema de tutoría fallará y se tuviera que reiniciar (véase el requisito no funcional de disponibilidad del tutor).

- **Fichero Mundo** (entrada). Este fichero, que se tratará de un fichero con extensión .txt, contendrá los objetos del entorno virtual que pueden ser bloqueados por los usuarios. Cada línea de este fichero contendrá el nombre de uno de estos objetos.

- **Fichero Npc** (entrada). Este fichero, también con extensión .txt, será un fichero auxiliar que con propósito de proporcionar ayuda a otra extensión externa, contendrá una serie de información de utilidad para esta otra extensión externa. Esto es debido a que será el tutor el que gestione todos los objetos utilizados por los avatares de los usuarios y por esta extensión. Dicha información consiste en: El identificador de la práctica, el nombre del objeto, acción que utilizará la extensión externa, coordenadas dentro del entorno virtual, orientación en forma de vector, todo ello separado por el carácter barra vertical (|). Esta información será utilizada por esta extensión externa para generar órdenes para un avatar automático o NPC (Non

Player Character) que se utilizará para enseñar a los estudiantes a llevar a cabo ciertas acciones de la práctica en el entorno virtual.

- **Mensaje de validación** (entrada). Los mensajes en este nuevo tutor procederán del entorno virtual en el que se haya desarrollado el laboratorio correspondiente, y que llegarán al sistema de tutoría a través de un módulo de comunicación que hará de intermediario entre dicho sistema y el entorno virtual. Al igual que en el antiguo sistema de tutoría, este mensaje tendrá el código de la acción que se desea validar. Es decir, el formato del mensaje será el mismo que en la versión anterior del tutor.

- **Mensaje de error de mundo** (entrada). A diferencia del sistema anterior, este nuevo sistema, que se encuentra desacoplado del motor gráfico, debe recibir por parte del entorno virtual en el que se encuentre desarrollado el laboratorio/práctica todos los errores de mundo ocurridos. Estos errores de mundo se refieren a aquellos errores que tienen que ver con que se cumplan o no ciertas condiciones geométricas relativas al avatar y a los objetos, como por ejemplo, que se intente dejar un objeto a mucha distancia de la mesa de turno, o que se pretenda tocar un objeto demasiado alejado del avatar. Estos mensajes deberán llevar consigo el nombre del objeto que produce el error así como el identificador del avatar o del usuario.

- **Mensaje de respuesta** (salida). Se trata del mensaje de respuesta del tutor una vez que este ha validado una acción recibida mediante un mensaje de validación. Dicha respuesta puede tratarse de un mensaje de aceptación, es decir, la acción ha sido validada sin ningún problema; un mensaje de tutoría, siendo este mensaje una pista para el usuario, la cual será la siguiente acción a realizar; o bien un mensaje de error debido a que el objeto está bloqueado, falta una o más acciones de las que depende, existe algún tipo de incompatibilidad o bien el tiempo asociado con la acción se ha cumplido.

4.2. Especificación no Funcional

Dentro de este apartado se detallarán aquellos requisitos que el nuevo sistema de tutoría satisfará y que no son requisitos funcionales. Dichos requisitos son los siguientes:

- **Portabilidad e independencia del motor gráfico**: Esto es, en cuanto a mejora se refiere, que el nuevo sistema de tutoría no se encontrará vinculado a ningún motor gráfico o entorno virtual como lo estaba el anterior sistema de tutoría, el cual se encontraba dentro de la plataforma de OpenSim. Esta independencia, que se ganará gracias a la implementación del nuevo sistema de tutoría como una librería externa escrita en el lenguaje C#, permitirá que el nuevo sistema de tutoría pueda anexarse a cualquier laboratorio desarrollado en cualquier motor gráfico (OpenSim, Unity 3D [Unity Technologies, 2014], etc.) como extensión de este, y la comunicación con este a través de cualquier medio de comunicación (llamadas a librería, cola de mensajes, etc.).

- **Integración con OpenSim**: El nuevo sistema de tutoría debe permitir una sencilla integración del mismo con la plataforma OpenSim. Dicha facilidad permitirá que el nuevo sistema de tutoría se integre con un coste irrisorio con el laboratorio de Biotecnología en el que se encontraba el anterior sistema.

- **Mayor eficiencia**: El nuevo sistema de tutoría tendrá una mayor eficiencia en comparación con el antiguo sistema. Esta eficiencia se reflejará a la hora de responder a las peticiones realizadas por los objetos al validar acciones llevadas a cabo por los usuarios, así como en la capacidad de trabajar con un mayor número de estudiantes al mismo tiempo (en el anterior sistema no se podían tener más de 5), cosa que el antiguo sistema de tutoría no permitía debido a su dependencia de la plataforma de OpenSim que no permitía un gran número de usuarios debido a los problemas presentes en esta plataforma de uso de memoria que terminaban por saturar esta y hacer que las respuestas fueran significantemente lentas para el usuario, llegando incluso a provocar colapsos del tutor.

- **Mejor mantenibilidad del código**: Debido a que el nuevo sistema de tutoría se implementará en el lenguaje orientado a objetos C#, se beneficiará de las ventajas que la programación orientada a objetos (POO) ofrece. El nuevo sistema hace uso de la POO para modularizar el código de tal manera que se puedan modificar los diferentes módulos sin que el sistema se vea comprometido a una gran reestructuración. Asimismo, un buen uso de mecanismos OO como pueden ser la herencia y el polimorfismo facilitará la extensibilidad y la futura reutilización del código. Y finalmente también se ha de remarcar la posible reutilización de multitud librerías ya existentes, como pueden ser el *framework* de .NET o librerías para el manejo de

ficheros de Microsoft Office (Excel, Word…), que ayudarán a reducir los costes de futuras extensiones o cambios.

- **Disponibilidad del tutor**: El sistema de tutoría además deberá ser capaz de poder manejar situaciones de caídas y recuperaciones de las mismas. Por ello, el tutor mantiene la persistencia de las trazas de los estudiantes en los ficheros asociados a la ontología del Estudiante. El contenido de estos ficheros se mantendrá sincronizado continuamente con la copia en memoria de la ontología. Gracias a esto, el tutor podrá recuperarse de posibles paradas forzadas por causas externas, tales como cortes de luz, caídas del servidor o máquina en el que esté alojado el laboratorio, fallos en la red de datos, etc. Dicha recuperación se llevará a cabo mediante la recarga en memoria de las trazas de los usuarios, permitiendo que el tutor pueda proseguir desde el último punto previo al fallo. Esto, sin embargo, no supone la recuperación del estado de los objetos dentro del motor gráfico, que en su defecto, se deberá solventar de manera independiente al tutor.

5. Desarrollo

En este apartado se detallará cómo se ha llevado a cabo el desarrollo del sistema automático de tutoría. Para ello se explicarán por separado los diferentes aspectos abordados, empezando por la arquitectura del sistema de tutoría y continuando con la descripción detallada de los paquetes que lo componen.

5.1. Arquitectura del Nuevo Tutor

El nuevo sistema automático de tutoría se estructura en una serie de paquetes con unas funciones específicas y que son independientes en un grado mínimo del resto de paquetes. A continuación se muestra el diagrama de paquetes que conforman la arquitectura del nuevo tutor:

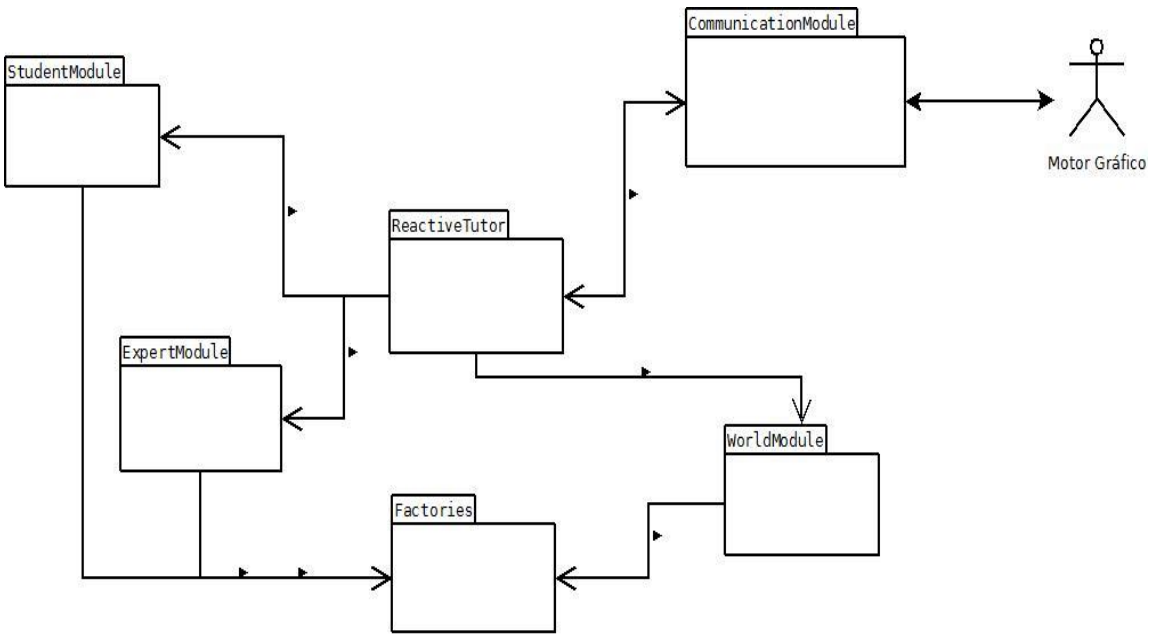

Figura 4.1. Diagrama de paquetes.

Como se puede apreciar en el diagrama, el agente central se trataría del paquete denominado *ReactiveTutor*, que sería el paquete donde se encontraría el bucle principal del tutor automático, cuyo comportamiento se detallará en un apartado posterior. En torno a este paquete central girarán el resto de paquetes que servirán de apoyo a la hora de realizar la tutoría automática. Cada uno de los paquetes realizará una función determinada dentro del proceso de tutoría en sí. Una vez obtenido el resultado de este

proceso de tutoría generado por el paquete central, *ReactiveTutor*, la respuesta se enviará al motor gráfico en donde se encuentre implementado el laboratorio virtual en cuestión a través del paquete denominado *CommunicationModule*.

Este paquete *CommunicationModule* será el paquete que se encargue de la integración del sistema automático de tutoría con el motor gráfico, así como de la comunicación entre ambas aplicaciones, el tutor y el laboratorio dentro del entorno virtual.

La integración del sistema con los laboratorios se realiza como se detalla a continuación:

- Laboratorio de Biotecnología (OpenSim): El tutor funcionará de manera externa a la plataforma y se comunicará con esta a través del módulo de comunicación del sistema de tutoría. Dentro de la plataforma y del laboratorio existirá un objeto primitivo que contendrá el *script* que recibirá los mensajes de los objetos como hacía el antiguo sistema. Dicho objeto se encargará de realizar las pertinentes llamadas al nuevo sistema de tutoría y recibirá la respuesta de este, mostrando al usuario el mensaje enviado por el tutor, sea de aprobación o indicando un error.
- Otros laboratorios (OpenSim): Con la implementación de otros laboratorios, el nuevo sistema recibirá los mensajes directamente de los objetos con los que los usuarios interaccionen a través de sus avatares. Es decir, los objetos que deban ser manipulados por los avatares, contendrán *scripts* que incluirán el código necesario para comunicarse directamente con el sistema de tutoría. Estos mismos objetos serán los que reciban la respuesta del tutor así como se encargarán de mostrar el mensaje recibido en dicha respuesta.
- Otros laboratorios (otros motores gráficos): La implementación en este caso será similar a la que se realizaría dentro de la plataforma de OpenSim. El paquete de comunicación en este caso deberá adaptarse al medio de comunicación que se elija para que el tutor y el motor gráfico puedan comunicarse. A su vez, el mundo desarrollado deberá contener el código necesario para poder comunicarse con el tutor, pudiendo estar este centralizado como en el caso del laboratorio de Biotecnología explicado un punto más arriba, o bien distribuido en el comportamiento de los objetos manejados por los avatares de los usuarios.

Para el caso específico que se trata en este trabajo, la reconstrucción del antiguo sistema automático de tutoría incluido en el laboratorio de Biotecnología dentro de la plataforma de OpenSim, y teniendo en cuenta que se pretende abarcar el mismo laboratorio, el paquete de comunicación se ha de implementar de forma que se pueda comunicar con la plataforma

de OpenSim. Para ello, dicho paquete deberá contener una clase que implemente una de las interfaces *ISharedRegionModule* o *INonSharedRegionModule* de la API de OpenSim[2]. Esta clase, denominada *RegionModule* dentro del entorno de OpenSim, es la clase que interactuará con la plataforma permitiendo el paso de información entre el tutor y el laboratorio. Esta clase deberá implementarse de una manera específica y además, deberá también tener una configuración especial para poder interactuar con la plataforma OpenSim[3].

Así mismo, algunos paquetes deberán acceder a la ontología del Estudiante para extraer o salvaguardar información en dicha ontología. El acceso a esta ontología se realizará a través de una clase contenida dentro del paquete denominado *StudentModule* y que se detallará más adelante. El acceso a esta ontología se hará por medio de la API Jena que proveerá los mecanismos necesarios para manejar todo el modelo ontológico así como manejar los datos que se encuentren almacenados en la ontología.

5.2. Descripción de los Paquetes

En este apartado se procederá a detallar exhaustivamente cada uno de los paquetes que componen la estructura del sistema automático de tutoría (véase Figura 4.1.). Cada uno de estos paquetes cumple una función dentro del sistema siendo independientes, en un grado mínimo, del resto de paquetes.

5.2.1. Paquete *ReactiveTutor*

Este paquete es el paquete que contendrá el funcionamiento e hilo principal de la aplicación. Será el encargado de realizar las pertinentes llamadas al resto de paquetes para poder responder a la acción recibida a través del paquete de comunicación. Este paquete posee la siguiente estructura:

[2] http://opensimulator.org/wiki/New_Region_Modules
[3] http://opensimulator.org/wiki/IRegionModule

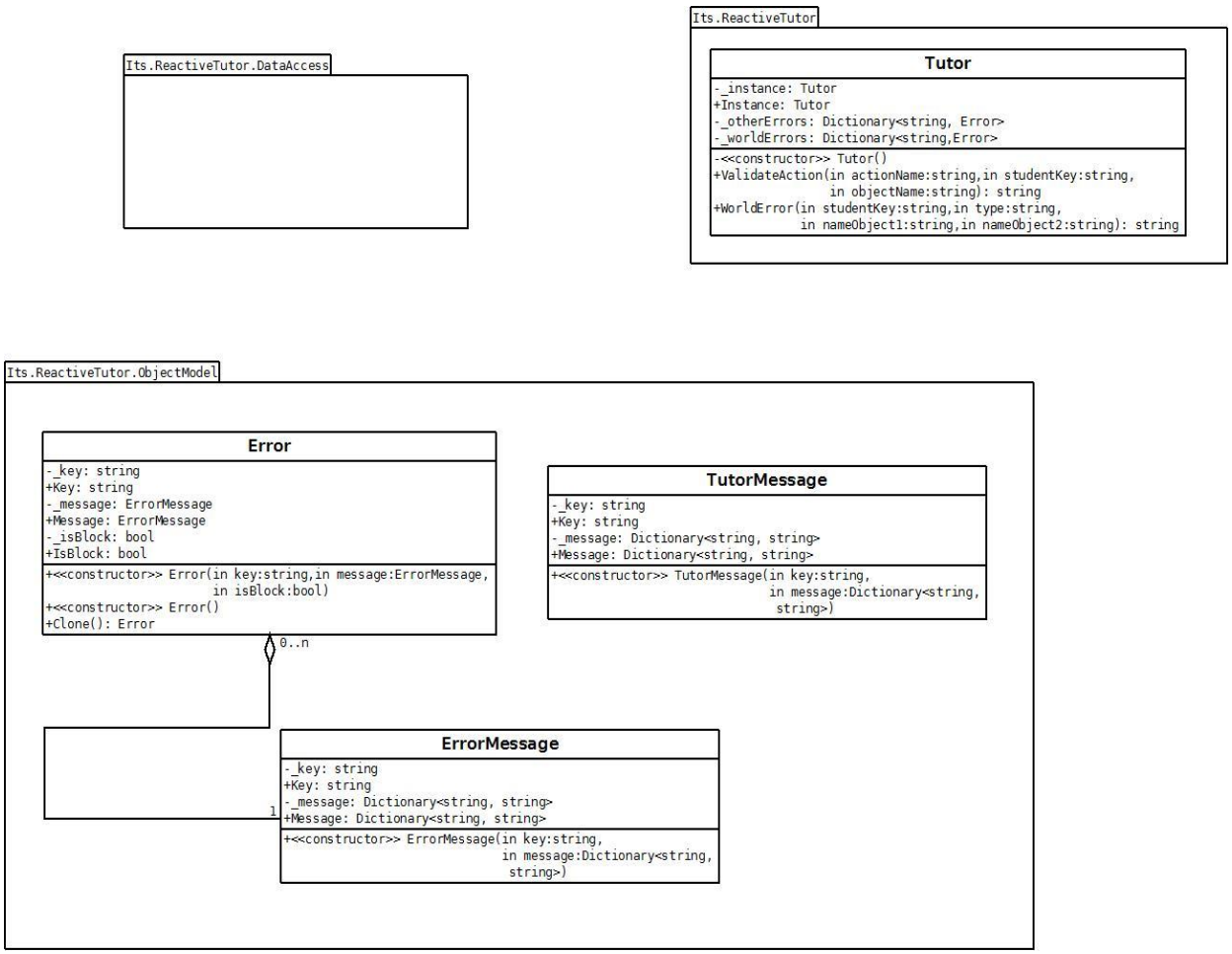

Figura 4.2. Diagrama de clases del paquete *ReactiveTutor*.

Visto el diagrama anterior, se puede observar que dentro del propio paquete *ReactiveTutor* existe una separación en paquetes internos que permiten separar las clases que lo componen según el uso que se dará de estas. Así pues, tendremos el paquete denominado *ObjectModel* que contendrá clases contenedoras de datos que servirán para manejar la información relativa a los errores que pueden cometer los usuarios y que el tutor deberá presentar a estos, así como los mensajes de errores y los de tutoría. También tendremos el paquete *DataAccess*, que se trata de un paquete que no posee clase alguna, pero que se encuentra presente con vistas al futuro uso de este paquete para albergar una clase que requiera el acceso a algún fichero que deba manejar el sistema automático de tutoría que será integrado en un sistema de tutoría más complejo. Por último, se tiene el paquete principal que contendrá la clase principal del sistema. A continuación, se procederá a explicar cuáles son las responsabilidades de cada clase dentro de los distintos paquetes:

- ***Error***: se trata de una clase contenedora, y en ella se almacena toda la información relativa a los errores que se pueden cometer durante la realización de la práctica. Esta información corresponde al mensaje de error que lleva asociado este y que se utilizará para dar noticia al usuario de que se ha cometido el error. Además posee un atributo que indica si este error, de cometerse, bloquearía el proceso y el usuario no podría continuar con la realización de la práctica. Por último, posee una clave única que lo identifica del resto de errores.
- ***ErrorMessage***: se trata de una clase contenedora que almacena un mensaje de error, es decir, el texto que el tutor deberá proporcionar cuando se produzca un error en concreto. Al igual que la clase anterior, posee una clave única que la diferencia e identifica de las demás instancias. El texto se almacena en un diccionario cuya clave para el almacenamiento del texto del mensaje será el código del idioma del mismo, ya que el texto del mensaje podrá encontrarse en distintos idiomas.
- ***TutorMessage***: Tal y como lo eran las dos clases previamente descritas, esta clase también se trata de una clase contenedora y en ella se almacenará un mensaje de tutoría, esto es, un mensaje que el tutor puede decidir dar al usuario como pista de cuál es la siguiente tarea que debe realizar dentro del protocolo de la práctica. Por lo tanto, esta clase contendrá un texto con dicho mensaje y una clave única que identifica y diferencia la instancia del resto de instancias de la clase. Al igual que ocurre con la clase anterior, los textos del mensaje puede encontrarse en varios idiomas, guardando estos en un diccionario utilizando como clave el código del idioma.
- ***Tutor***: Esta es la clase principal del paquete y de la aplicación. Será la encargada de realizar las pertinentes llamadas a las clases de apoyo contenidas en los otros paquetes para llevar a cabo el proceso de tutoría. La clase contiene dos contenedores en los que se almacenarán los errores que se pueden producir en el mundo y un tipo especial de errores relacionados con problemas como pueden ser el intento de repetir una acción que no está permitida repetir o bien el intento de realizar una acción que no se encuentra en la configuración de la práctica. Para el funcionamiento de la acción de tutoría en sí, la clase poseerá dos métodos:

    - *ValidateAction*: Es el método que se encargará de realizar las llamadas necesarias para la validación de una acción tal y como se realizaba en el antiguo tutor. Al igual que este, recibirá como parámetro el nombre de la acción y el objeto que se ha utilizado, y con las llamadas pertinentes, se validará la acción y se bloqueará o no el uso del objeto. Como respuesta,

el tutor devolverá una cadena de caracteres, la cual será el mensaje de respuesta a la acción de tutoría, es decir, el mensaje devuelto será un mensaje de confirmación, con o sin mensaje de tutoría, o un mensaje de error, este último dependiendo de la configuración del mensaje.

- *WorldError*: Se trata de un método que registrará dentro del sistema de tutoría los errores de mundo ocurridos dentro del entorno virtual. Estos errores de mundo son errores tales como puede ser el relacionado con que un usuario quiera tocar un objeto estando situado a una excesiva distancia de este, o que se pretenda dejar un objeto en un sitio que no le corresponde. Como estos errores son detectados dentro del motor gráfico utilizado, y la aparición de uno de estos errores se debe registrar dentro del tutor, esta función se encargará de ello. Estos errores de mundo están acotados a un número exacto de errores, pudiendo estos variar tan solo en los objetos que forman parte en ellos, por lo tanto, se han definido dentro de la ontología del estudiante unos mensajes genéricos para estos errores y que, como se describirá más adelante, serán manejados por el paquete *WorldModule*. Será entonces cometido de este método comprobar que dicho error de mundo se encuentra dentro de los contenedores y devolver el mensaje asociado a este o realizar una llamada al paquete *WorldModule* para que cree un nuevo error de mundo específico en la ontología. Así mismo, devolverá el mensaje de error necesario para mostrar al usuario.

Otra responsabilidad de esta clase, asignada al constructor de la misma, será la de detectar si se está produciendo la recuperación de una posible caída del programa, esto es, la clase deberá comprobar cuando se inicialice el programa si dicha inicialización se debe a que se está volviendo a ejecutar el programa tras una caída o se trata de la ejecución desde cero de un nuevo laboratorio.

5.2.2. Paquete *ExpertModule*

Este paquete se encargará de todo lo relacionado con el protocolo de la práctica, es decir, la lista de acciones que se deben realizar para completar la práctica y que componen a esta. Por consiguiente, este paquete contendrá la información relacionada con las acciones, así como la validación y ejecución de estas. Al igual que el anterior paquete, se compone de pequeños paquetes, teniendo la siguiente estructura:

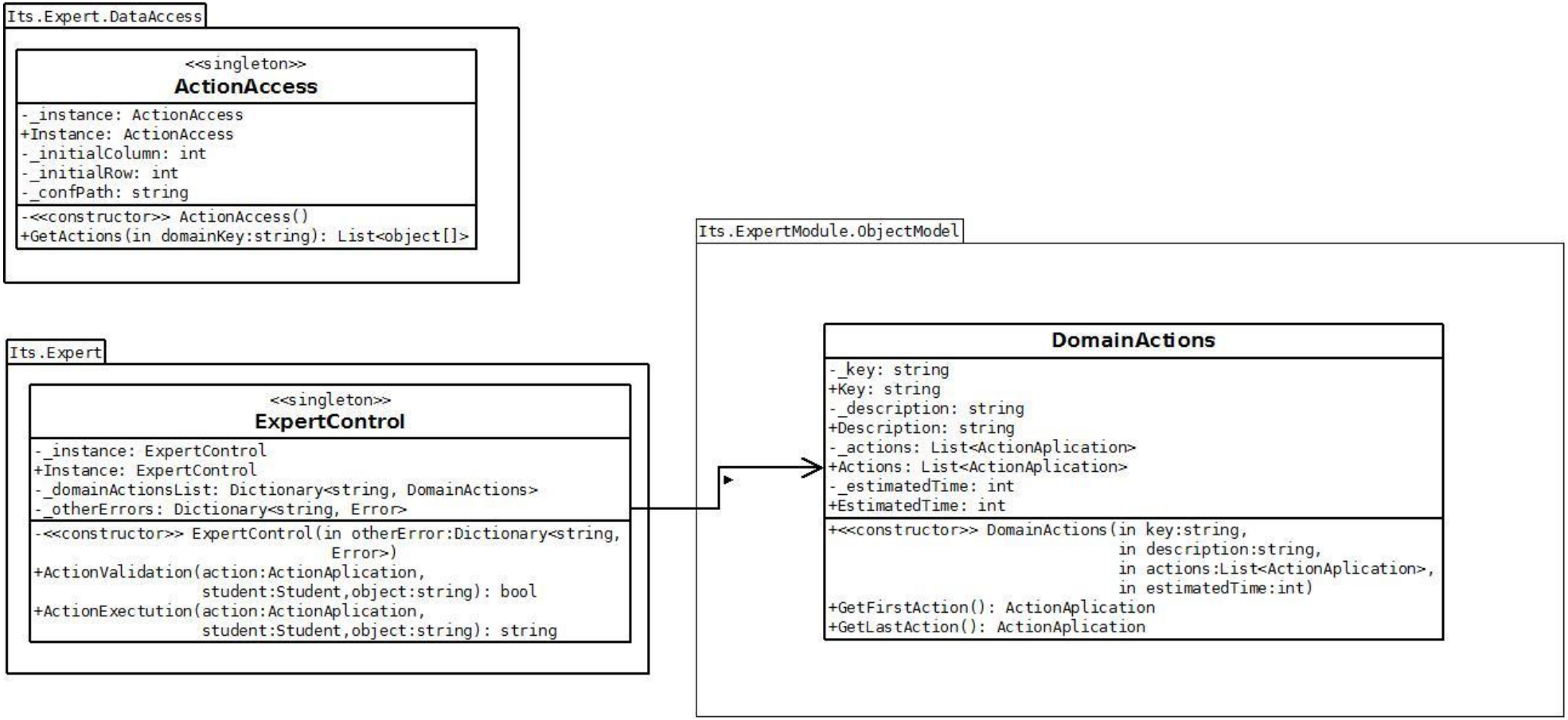

Figura 4.3. Diagrama de clases del paquete *ExpertModule*.

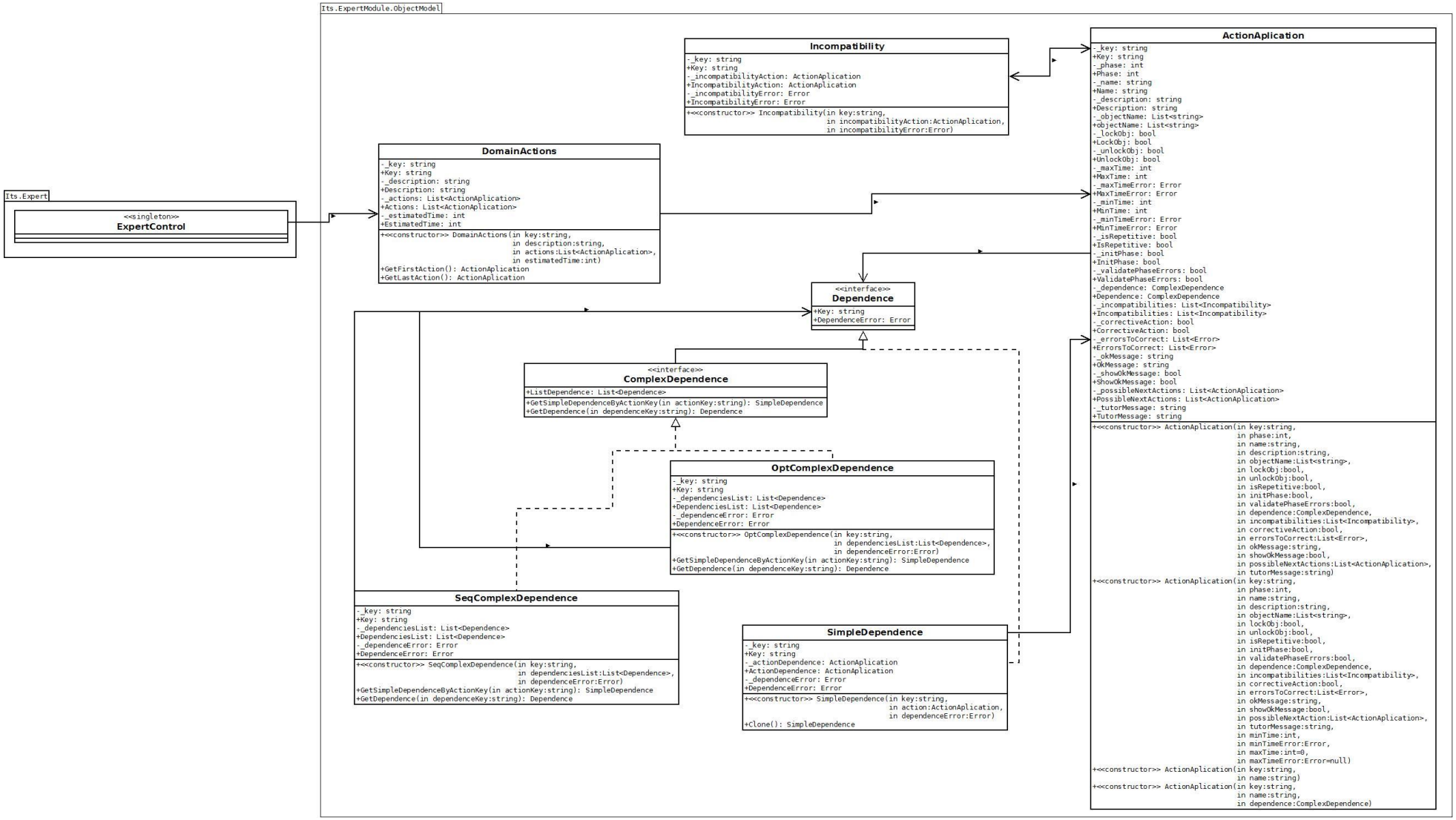

Figura 4.4. Diagrama de clases del paquete *ExpertModule*, continuación.

Al igual que el paquete *ReactiveTutor*, este paquete también se subdivide en los paquetes: *DataAccess*, que es el paquete que tiene acceso al fichero de configuración de la tutoría (véase apartado 3.1.2. Especificación de las Entradas y Salidas) y que extraerá la información de dicho fichero para la creación de las acciones por parte de las factorías, lo cual se explicará posteriormente; *ObjectModule*, que contiene la definición de las clases contenedoras en donde se albergará toda la información requerida por el paquete para la ejecución del módulo; y *Expert*, que contendrá la clase principal del paquete encargada de manejar la información relacionada con el protocolo de la práctica. Las clases presentes en el paquete *ExpertModule* son:

- ***ActionAccess***: Esta clase será la encargada, mediante la utilización de una librería que permite la lectura de ficheros con extensión .xlsx, de leer la información contenida en los ficheros de configuración de la tutoría de cada una de las acciones que compone la práctica. Esta información se obtiene a través del método *GetActions*, que se encargará de leer la información del fichero, devolviendo una lista con un elemento con los datos de cada acción por cada fila existente dentro del fichero de configuración.
- ***DomainActions***: Esta clase es la clase contenedora de la configuración de la práctica. En ella se almacenarán todas las acciones que conforman la práctica, así como la información de la misma como son el nombre, la descripción de esta y el tiempo estimado que se tarda en realizar.
- ***ActionAplication***: Esta clase contenedora de datos tendrá toda la información relacionada con una acción de la práctica. Esta información será de vital importancia a la hora de realizar la tutoría, y será la información contra la que interactuará la clase principal del paquete para realizar la validación de una acción enviada por parte del paquete *ReactiveTutor*. Cada instancia se corresponderá con una línea del fichero de configuración de la tutoría.
- ***Incompatibility***: Se trata de la clase que contendrá la información relativa a las incompatibilidades que puede presentar una acción. En ella se almacenará la acción que es incompatible y el error asociado a dicha incompatibilidad, que será el error que contenga el mensaje que el tutor deberá dar al usuario de ocurrir dicha incompatibilidad.
- ***Dependence***: Interfaz que representa las dependencias de una acción.
- ***ComplexDependence***: Interfaz que hereda de la interfaz *Dependence*, y que representa dependencias complejas. Estas dependencias poseerán una o más dependencias simples (véase a continuación).
- ***SimpleDependence***: Clase que implementa la interfaz *Dependence* y que representa, lo que se ha denominado, dependencia simple.

Esta dependencia contiene una única acción de la que es dependiente, así como el error asociado a dicha dependencia.

- ***OptComplexDependence***: Clase que implementa la interfaz *ComplexDependence* y que representa las denominadas, dependencias opcionales. Esta clase de dependencias contienen una o varias dependencias simples cuyas acciones pueden ejecutarse en cualquier orden, siempre que se ejecuten todas. También posee el error asociado con esta dependencia que será utilizado en el caso de que alguna de las acciones de las dependencias no haya sido ejecutada.
- ***SeqComplexDependence***: Clase que implementa la interfaz *ComplexDependence* y que representa las denominadas, dependencias secuenciales. Esta clase de dependencias requieren la ejecución de las acciones de las dependencias simples que la componen en el mismo orden en el que se encuentran dentro de la lista. El error asociado será aquel que será utilizado en caso de que no se haya realizado la secuencia de acciones determinada por la lista de dependencias simples.
- ***ExpertControl***: Se trata de la clase principal del paquete y será la encargada de ejecutar y validar las acciones que reciba el tutor. Esta clase contendrá todos los *DomainActions* de las prácticas y todas las *ActionAplication* incluidos en estos. Tendrá entonces dos métodos para cada una de las dos posibles operaciones, validar y ejecutar una acción. A la hora de validar una acción esta clase se comunicará con el paquete *StudentModule* que contendrá las trazas de los estudiantes y, con arreglo a la información de la *ActionAplication* a validar y la traza del estudiante que ha intentado ejecutar la acción, dará como validada o no la acción, respondiendo con un valor booleano al tutor. En el caso de que la acción sea validada, será necesario llamar al método que ejecutará la acción, que básicamente consistirá en guardar una traza con la información necesaria, que se almacenará en la ontología del Estudiante.

5.2.3. Paquete *Factories*

Este paquete contendrá las clases factorías de objetos, es decir, las clases encargadas de crear las instancias de los contenedores de datos utilizados por los distintos paquetes. La estructura del paquete es la siguiente:

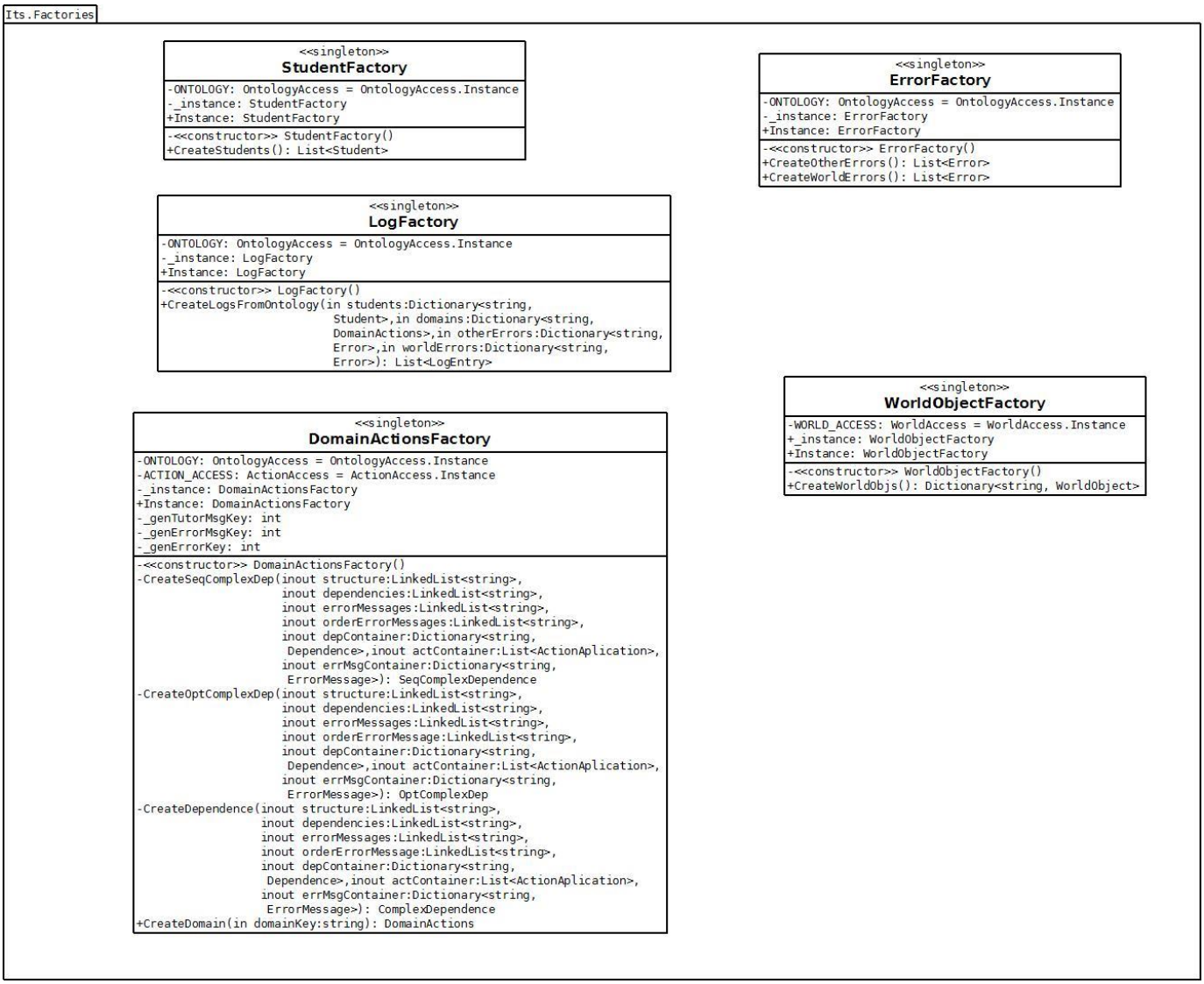

Figura 4.5. Diagrama de clases del paquete *Factories*.

Este paquete es especial con respecto al resto debido a que este paquete no se subdivide en otros paquetes, tan solo contiene las clases factorías. Estas clases tienen las siguientes funciones:

- ***StudentFactory***: Esta factoría se encargará de crear las instancias de la clase *Student* (véase más adelante en el paquete *StudentModule*) a partir de las fichas técnicas de los estudiantes que se encuentran almacenadas dentro de la ontología del Estudiante. Para ello, hará uso de una clase contenida dentro del paquete *StudentModule* que tiene acceso a la ontología y de ahí obtendrá la información. Obtenida la información de todos los estudiantes, la factoría creará todas las instancias y las devolverá como una lista.
- ***ErrorFactory***: Esta clase se encarga de obtener los mensajes de error genéricos para aquellos errores de mundo o errores "otros" que

se encuentran en la ontología del Estudiante. Hará uso de la misma clase que la factoría anterior y obtendrá todos los mensajes. Para la creación de unos u otros mensajes se hará uso de los métodos al respecto, un método para los errores de mundo y otro método para los errores "otros". Estos métodos devuelven listas con los errores extraídos de la ontología.

- ***WorldObjectFactory***: Esta factoría se encargará de la creación masiva de las instancias de la clase *WorldObject* contenida en el paquete *WorldModule*, explicado en un apartado posterior. La información necesaria para la creación de estas instancias se obtiene del fichero Mundo (véase el apartado 3.1.2. Especificación de las Entradas y salidas), de donde obtendrá la información de los objetos del mundo de cada una de las líneas que componen el fichero. Esta información la proveerá la clase *WorldAccess*, del paquete *WorldModule*, clase que se explicará en el apartado correspondiente. Una vez obtenida toda la información, la factoría creará las instancias y las devolverá como un diccionario, donde la clave es el nombre identificativo del objeto del mundo.
- ***LogFactory***: Esta clase se encargará de la creación de las instancias de los *logs* de las trazas de los estudiantes guardados en la ontología del Estudiante. Esta clase será utilizada por el tutor para recuperar todos los *logs* guardados tras una caída del tutor. Para ello utilizará la clase encargada del acceso a la ontología del Estudiante y obtendrá la información de los alumnos pasados como parámetro para un determinado *DomainActions* o práctica. Tras obtener la información de la ontología, la clase se encargará de crear los distintos *logs* y los devolverá como una lista.
- ***DomainActionsFactory***: Es la clase encargada de la creación de los *DomainActions* con todas sus *ActionAplication* a partir de los ficheros de configuración de la tutoría. Para ello, se hará uso del método público que se encargará de crear el *DomainActions* con la clave de práctica pasada como parámetro. Este se valdrá de la clase *ActionAccess*, explicada anteriormente, para acceder al fichero de configuración de la tutoría para la práctica deseada y obtendrá toda la información de las filas del fichero .xlsx. Tras esto, creará las diferentes clases y sus atributos a partir de dicha información. Un aspecto importante dentro de las clases creadas son las dependencias de las *ActionAplication* que, por su complejidad, la factoría se valdrá de un grupo de métodos internos para la creación de estas. Estos métodos se utilizarán siguiendo un algoritmo recursivo para la creación de la estructura de las dependencias. Este algoritmo tiene el siguiente funcionamiento:

```
CreateComplexDependence () {
        If nextElement() = [
        Then CreateOptComplexDependence()
        Else if nextElement() = {
        Then CreateSeqComplexDependence()
}

CreateSeqComplexDependence () {
        new list
        While (hasNextElement)
                If nextElement() = [
                Then list.add(CreateOptComplexDependence())
                Else if nextElement() = {
                Then list.add(CreateSeqComplexDependence())
                Else if nextElement() = actionId
                Then list.add(new SimpleDependence(actionId))
                Else if nextElement() = ]
                Then exception()
                Else if nextElement() = }
                Return list
        If lastElement() != }
        Then exception()
}

CreateOptComplexDependence() {
        new list
        While (hasNextElement)
                If nextElement() = [
                Then list.add(CreateOptComplexDependence())
                Else if nextElement() = {
                Then list.add(CreateSeqComplexDependence())
                Else if nextElement() = actionId
                Then list.add(new SimpleDependence(actionId))
                Else if nextElement() = }
                Then exception()
                Else if nextElement() = ]
                Return list
        If lastElement() != ]
        Then exception()
}
```

Figura 4.6. Pseudocódigo del algoritmo de creación de dependencias.

Tras la ejecución de este algoritmo, y en el caso de no haber ocurrido ningún error, se habrá obtenido la estructura de dependencias de la *ActionAplication*. En caso de haber ocurrido un error, supondrá que la estructura de dependencias presente en el fichero de configuración de la tutoría está mal construida, y por lo tanto, dicho fichero deberá ser revisado y corregido.

Una vez obtenidas todas las *ActionAplication* con sus respectivos atributos, se creará el *DomainActions* correspondiente. Tras crearse el valor a devolver, tanto el *DomainActions* como las *ActionsAplication* creadas serán guardadas dentro de la ontología del Estudiante para su futura utilización a la hora de crear los *logs* de los estudiantes. Para ello se utilizará la clase de acceso a la ontología que guardará en esta toda la información necesaria para representar las trazas dentro de la ontología. Al final de la ejecución, la clase devolverá la *DomainActions* solicitada.

5.2.4. Paquete *WorldModule*

Este es el paquete encargado de manejar toda la información del mundo necesaria por el sistema automático de tutoría. Esta información tiene que ver con los objetos que pueden ser bloqueados por los estudiantes y que por lo tanto, no podrán ser usados por otros. El tutor deberá conocer dichos bloqueos para realizar las pertinentes acciones de tutoría. Además, también contendrá la información necesaria para proveérsela a una extensión de manejo de un NPC (Non Player Character), que al igual que el sistema automático de tutoría, estará integrado en un sistema de tutoría más complejo. Este paquete posee la siguiente estructura, también subdividida en paquetes más pequeños que separan las funcionalidades de las clases:

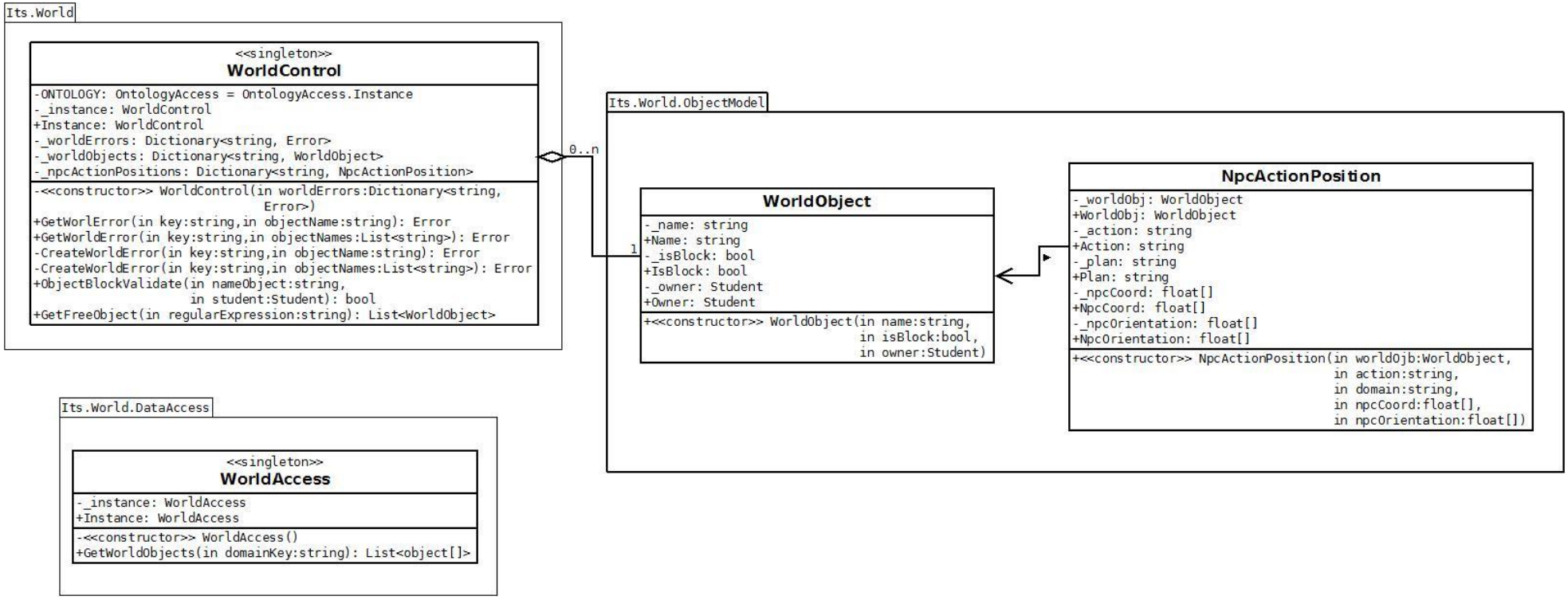

Figura 4.7. Diagrama de clases del paquete *WorldModule*.

Teniendo en cuenta las clases presentes en el diagrama anterior, a continuación se procede a explicar cuáles son las funciones de cada clase:

- ***NpcActionPosition***: Esta clase es una clase contenedora de información, y la información contenida es la que se proveerá a la extensión del manejo de un NPC. La información más importante almacenada son las coordenadas donde se situaría el NPC a la hora de ejecutar una determinada acción, definida dentro de la clase, y la orientación que tendrá que tener el NPC.
- ***WorldObject***: Esta clase contenedora es la que poseerá la información relativa a los objetos dentro del mundo con los que interactuarán los usuarios. Esta información corresponde con el nombre del objeto, si se encuentra este bloqueado y el estudiante que lo está bloqueando, porque lo está utilizando.
- ***WorldAccess***: Esta clase será la encargada del acceso a los ficheros que contendrán la información de los objetos bloqueables de las prácticas. Esta clase deberá estar implementada con arreglo al tipo de fichero que se maneje, por lo que está ligada a los ficheros que manejen los objetos. Para la obtención de dicha información, se deberá proceder al uso del método correspondiente y este devolverá la información como una lista de arrays como elementos de la lista, los cuales tendrán la información de cada objeto con un determinado formato.
- ***WorldControl***: Esta clase es la principal del paquete y se encarga del manejo de los objetos así como de la creación y almacenamiento de los errores del mundo. Además, también será la clase a la que la extensión del manejo de un NPC deberá acudir para obtener la información requerida por esta. La clase posee una serie de contenedores que almacenan los errores de mundo, los objetos y las posiciones del NPC. Las posibles operaciones que ejecutará esta clase son:

    - Validar el bloqueo de un objeto. Esta función consistirá en comprobar si un objeto dado se encuentra bloqueado por un determinado estudiante. Esta comprobación buscará la instancia de *WorldObject* y comprobará si está bloqueado o no. Los casos serían:

        a) Que se encuentre el objeto bloqueado, y por lo tanto habría que comprobar si es por el mismo estudiante o por otro estudiante. En caso de ser por otro estudiante, el valor devuelto será un valor *false* booleano, mientras que será un valor *true* en caso de que sea por el mismo

usuario. En caso de devolverse el primer valor, aquella clase que hubiera realizado la llamada al método, debería de encargarse de notificar al usuario.

b) Que no se encuentre el objeto bloqueado. En este caso, la clase bloquearía el objeto con el estudiante pasado como parámetro como nuevo propietario del objeto.

- Obtener la información del NPC para una acción. Esto proveerá la posición y la orientación que necesita la extensión para situar al NPC dada una expresión regular determinada pasada como parámetro. A partir de la expresión se deducirá qué objeto se solicita y qué acción se está ejecutando, por lo que la clase buscará la instancia de *NpcActionPosition* determinada y devolverá la información solicitada.

- Obtener errores de mundo. Dados la clave del error y el/los nombre/s del/de los objeto/s implicados. Si se da el caso de que el error solicitado no existiera, este método se encargaría de crear el error del mundo del tipo solicitado con los objetos pasados como parámetro, devolviendo el error resultante.

5.2.5. Paquete *StudentModule*

Este paquete será el encargado de todo lo relacionado con el estudiante, la ontología del Estudiante y el control y salvaguarda de las trazas de los estudiantes. Al igual que los paquetes *ReactiveTutor*, *ExpertModule* y *WorldModule*, este paquete se subdivide en tres paquetes para la separación de las clases según su función. Por lo tanto, la estructura del paquete será la siguiente:

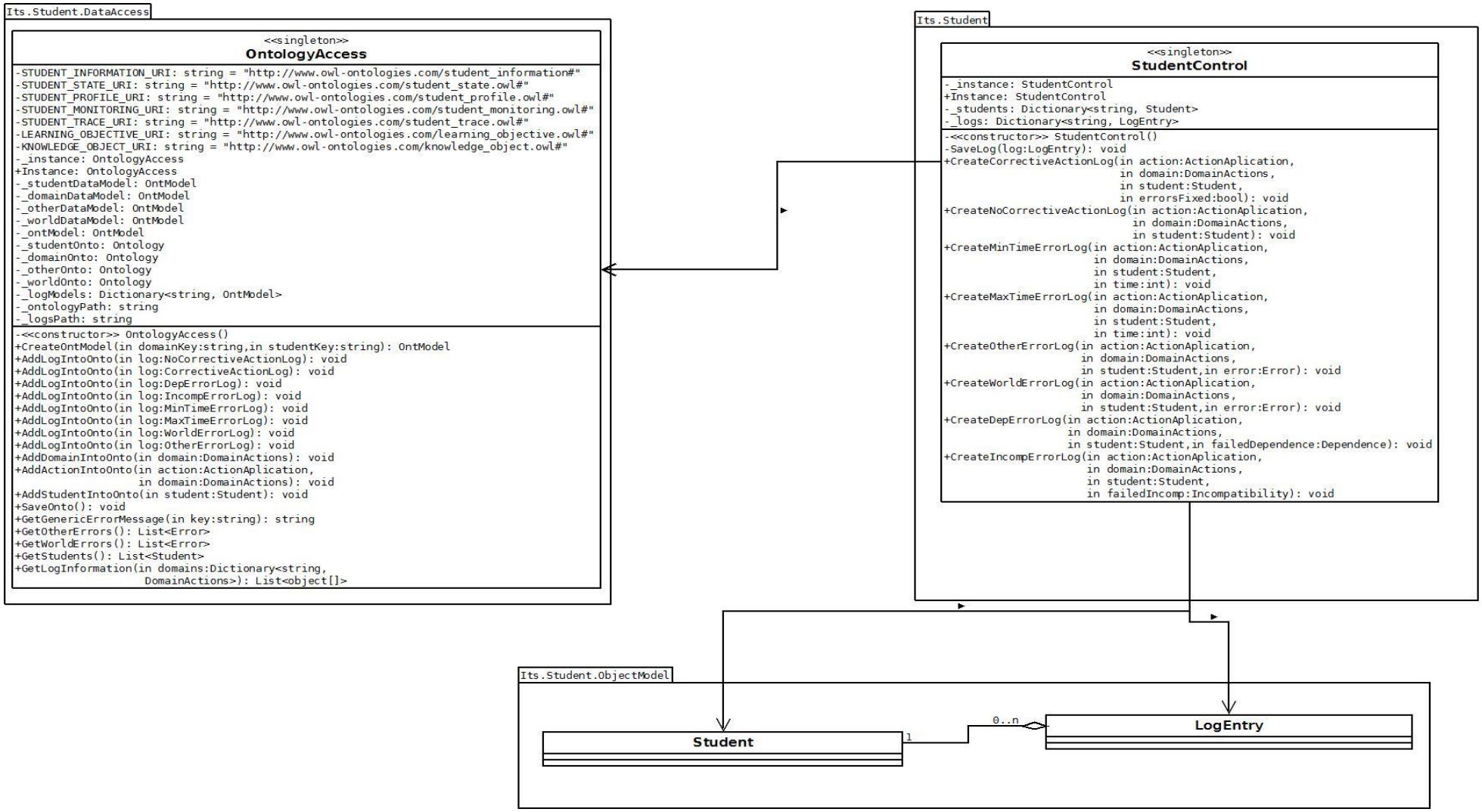

Figura 4.8. Diagrama de clases del paquete *StudentModule*.

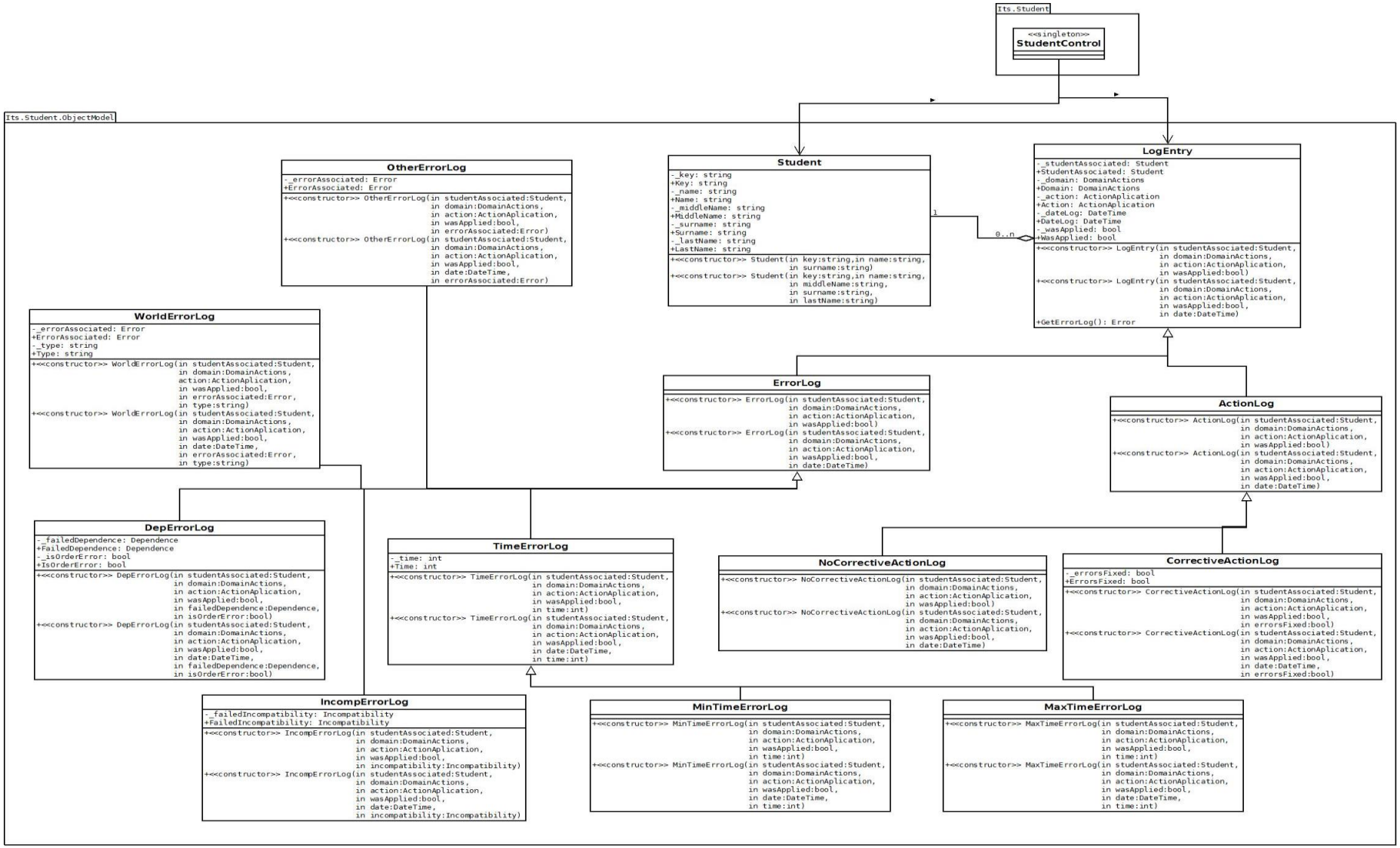

Figura 4.9. Diagrama de clases del paquete *StudentModule*, continuación.

Cada una de las clases tiene una función determinada, que se explica a continuación:

- ***Student***: Es la clase contenedora de la información relativa al estudiante. En ella se almacena información como el nombre del estudiante, los apellidos y el segundo nombre, de tenerlo. Además, posee una clave única que identifica la instancia del resto de instancias.
- ***LogEntry***: Clase que representa los *logs* de los estudiantes. De esta clase heredan los distintos tipos de *logs* que se pueden dar según las acciones y errores cometidos por los estudiantes. Almacena la información de un intento de acción que se pretende registrar en la traza, el dominio de la práctica, el estudiante que la realizó, la fecha y hora en la que se realizó y un valor booleano que representa si la acción se ejecutó o no finalmente.
- ***ActionLog***: Hereda de la clase anterior, y representa los *logs* de las acciones que se han realizado correctamente. De esta clase heredan los *logs* que representan las acciones realizadas correctamente.
- ***NoCorrectiveActionLog***: Esta clase hereda de la clase *ActionLog* y representa aquellas acciones realizadas correctamente pero que no corrigen errores. Esto es, debido a la integración del sistema automático de tutoría en un sistema de tutoría mayor, es posible que algunas acciones puedan corregir los errores cometidos por los estudiantes. Por lo tanto, esta clase representa las acciones realizadas correctamente pero que no corrigen errores.
- ***CorrectiveActionLog***: Esta clase, al igual que la anterior, hereda de la clase *ActionLog* y representa aquellas acciones realizadas correctamente que son acciones que han corregido algún error o todos los existentes hasta entonces. Para ello, posee un atributo que indica si la acción ha corregido todos los errores o no, tomando valor *false* en caso de no haber corregido todos los errores, pero sí uno o varios de ellos, y toma valor *true* si la acción ha corregido todos los errores existentes.
- ***ErrorLog***: Esta clase representa los *logs* que guardan las acciones realizadas incorrectamente por el estudiante. Hereda de la clase *LogEntry* y de ella heredan todas las clases que representan los diferentes tipos *logs* de errores.
- ***OtherErrorLog***: Esta clase representa los *logs* causados por errores del tipo "otros", es decir, para errores cuando ya se han realizado acciones repetidas o cuando no se ha encontrado una acción determinada en una fase de la práctica. Tiene asociado el error correspondiente en forma de atributo interno.
- ***WorldErrorLog***: Clase que hereda de *ErrorLog*, representa los *logs* que contienen los errores de mundo. Poseen como atributos internos

el error de mundo ocurrido así como el tipo de error de mundo que es (véase más adelante tipos de errores de mundo).

- ***DepErrorLog***: Representa los *logs* de errores por dependencias. En ellos se almacena toda la información referente a los *logs* heredada de la clase *LogEntry* y la dependencia que causó el error, además de un valor que indica si el error fue por el orden de las dependencias o no.
- ***IncompErrorLog***: Clase que contiene la información referente a los *logs* que causados por errores por incompatibilidades. Posee un atributo que almacena la incompatibilidad causante del error.
- ***TimeErrorLog***: Clase que almacena la información de los *logs* por errores de tiempo. De esta clase heredan las clases de tipo que representan los *logs* por exceder el tiempo máximo permitido o por no esperar el tiempo mínimo requerido. Por lo tanto, tendrá como atributo el tiempo que cause el error.
- ***MinTimeErrorLog***: Representa los *logs* de errores por no haber cumplido el mínimo tiempo requerido.
- ***MaxTimeErrorLog***: Representa los *logs* de errores por haber excedido el tiempo máximo expresado.
- ***StudentControl***: Esta clase es la clase central del paquete y se encarga del manejo de toda la información de los estudiantes y de los logs de estos. Por lo tanto, como manejador, poseerá unas estructuras de datos que almacenarán a los estudiantes y a los logs por estudiantes. Además, es la encargada de crear los *logs* mediante las pertinentes llamadas a los diferentes métodos para crear los distintos tipos de *logs*, así como es la encargada de guardar estos en la ontología mediante el uso de la clase *OntologyAccess*.
- ***OntologyAccess***: Es la clase encargada de manejar los ficheros y el modelo ontológico durante la ejecución del sistema de tutoría. Como clase de acceso, tiene la responsabilidad de guardar los *logs* en la ontología del Estudiante, así como de recuperar la información de estos desde la ontología. También tiene que ser capaz de recuperar los errores de los tipos "otros" y de mundo, así como la información de los estudiantes. El modelo ontológico se guarda en unos ficheros con extensión .owl. Estos ficheros se encontrarán en una carpeta dentro del directorio de instalación del sistema automático de tutoría y tendrá la siguiente estructura:

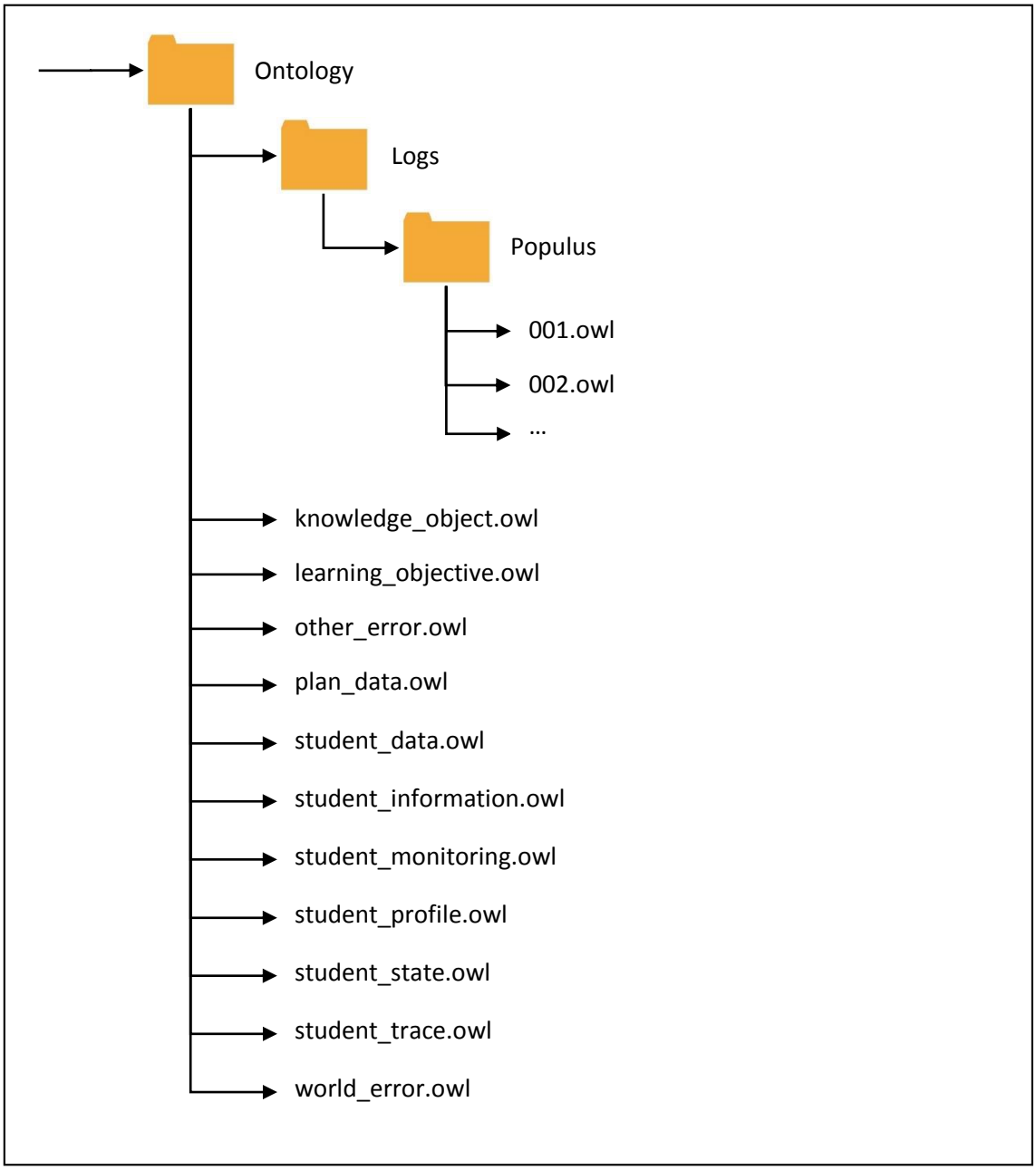

Figura 4.10. Jerarquía de ficheros de la ontología del Estudiante.

En la figura anterior se muestra la jerarquía que compondrán todos los ficheros que formen la ontología del Estudiante. Estos ficheros poseen una función específica que se detallará a continuación:

- knowledge_object.owl, learning_objective.owl, student_information.owl, student_monitoring.owl, student_profile.owl, student_state.owl y student_trace.owl conforman los ficheros propios de la ontología del Estudiante y del modelo ontológico [Clemente et al., 2011].

- other_error.owl, será el fichero donde se almacenen los mensajes genéricos para los errores del tipo Other. Esta información será necesaria a la hora completar los *logs* cuando éstos se guarden en la ontología del Estudiante.
- plan_data.owl, será el fichero donde se guarde la información de la práctica así como las acciones que la conforman. Esta información será necesaria a la hora completar los *logs* cuando estos se guarden en la ontología del Estudiante.
- student_data.owl, será el fichero donde se guarde toda la información de los estudiantes. Desde este fichero se recuperará la información para crear las instancias de la clase *Student*.
- world_error.owl, será el fichero donde se almacenen los mensajes para los errores de mundo. En él se encontrarán tanto los mensajes genéricos para los distintos tipos de errores de mundo, como los mensajes específicos para determinados errores producidos por ciertos objetos.
- Logs, será la carpeta donde se almacenen los *logs* de los estudiantes. Esta carpeta contendrá, a su vez, una carpeta por cada práctica que maneje el sistema automático de tutoría. Como se puede ver en la Figura 4.8, existe una carpeta para una práctica llamada Populus. Dentro de cada carpeta de práctica se encontrará un archivo con extensión .owl donde se guarden los *logs* de cada estudiante con el identificador que da nombre al fichero. En el ejemplo de la Figura 4.8, existen dos ficheros, el del estudiante con identificador 001 y otro para el estudiante con identificador 002, ficheros 001.owl y 002.owl respectivamente.

5.2.6. Paquete *CommunicationModule*

Este es el paquete que manejará la comunicación entre el sistema automático de tutoría y el motor gráfico. Es totalmente dependiente del motor gráfico y su implementación dependerá del método de comunicación que se tenga entre el tutor y el motor. Dado que este trabajo fin de grado está orientado a la implementación del sistema automático de tutoría para la plataforma de OpenSim, la implementación se ha realizado para esta. Por otro lado, esta implementación dependerá de cuál de las dos integraciones se fuera a llevar a cabo (véase apartado 4.1. Arquitectura del Nuevo Tutor), pero ya que ambos métodos de integración deberán implementar las interfaces *ISharedRegionModule* y *INonSharedRegionModule* de la API de

OpenSim[4], así como la determinada configuración para la implementación del *IRegionModule*[5], el paquete *CommunicationModule* será el mismo para ambos casos.

Teniendo en cuenta que la característica que diferencia a una y otra implementación es la de la comunicación a través de un objeto central como es el caso del laboratorio de Biotecnología, o directamente desde cada objeto, como es el caso de los nuevos laboratorios, y esto es en ambos casos dependiente de la plataforma, la clase encargada de la comunicación se implementará de la misma forma para ambas integraciones. Por lo tanto, la estructura del paquete, para ambas integraciones, será la que sigue:

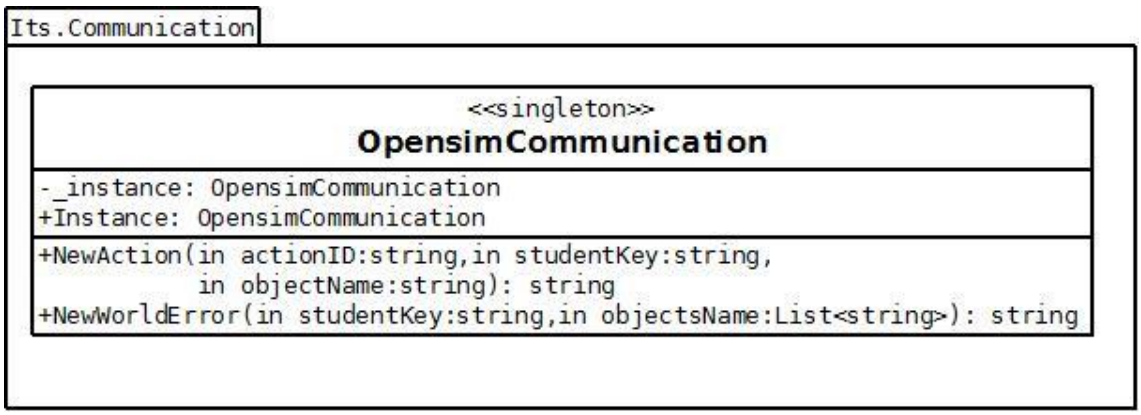

Figura 4.11. Diagrama de clases del paquete *CommunicationModule*.

A diferencia de otros paquetes, este solo contiene una única clase, que será intermediadora entre el sistema automático de tutoría y el motor gráfico donde se encuentre implementado la práctica. Esta clase deberá implementar la interfaz *IRegionModule* para que, desde OpenSim, se puedan hacer llamadas a los métodos que en la clase se definan. Para el sistema automático de tutoría, se han definido dos métodos:

- NewAction(). Al cual se deberá de llamar desde la plataforma de OpenSim para la validación y ejecución de una acción realizada por el avatar de un estudiante. Este método se encargará de llamar al tutor para la validación de la acción y obtendrá de esta llamada el mensaje que deberá de enviar al usuario.

- NewWorldError(). Será el método al que se llamará desde la plataforma en caso de ocurrir un error en el mundo. Este método llamará al método correspondiente del tutor para registrar el mundo

[4] http://opensimulator.org/wiki/New_Region_Modules
[5] http://opensimulator.org/wiki/IRegionModule

y generar el correspondiente *log* que deberá ser guardado en la ontología del Estudiante.

Esta clase, por lo tanto, será la utilizada para la comunicación específica con la plataforma de OpenSim, y válida tanto para el laboratorio de Biotecnología como para los nuevos laboratorios que se implementen y quieran hacer uso del sistema automático de tutoría. Para el caso de otras plataformas, este paquete y esta clase, se deberán adecuar al motor gráfico que se utilice.

6. Conclusiones y Trabajo Futuro

Como conclusiones se puede destacar que la nueva estructura del sistema automático de tutoría que en este trabajo fin de grado se ha desarrollado supondrá una mejora en cuanto a eficiencia en tiempo de ejecución con respecto al anterior sistema de tutoría, gracias a la utilización de la programación orientada a objetos así como la utilización de objetos en memoria, en vez del manejo de ficheros (*notecards*) que utilizaba el anterior tutor. Además, el nuevo diseño abre la posibilidad de usar el sistema de tutoría fuera de la plataforma de OpenSim, haciéndolo portable a otras plataformas y motores gráficos. Esto se ha conseguido gracias a la codificación del mismo como una librería escrita en el lenguaje C#, que permite su uso en distintos sistemas operativos, todo ello sin renunciar a la posibilidad de seguir utilizándolo en la plataforma OpenSim. Por otro lado, el nuevo tutor posee una mejor mantenibilidad del código gracias a su estructuración más modular mediante paquetes y clases, y en la que su legibilidad y organización es mejor que la presente en el código del anterior tutor. Como consecuencia de esta estructura más modular, es posible modificar ciertas partes del código sin que se vean afectadas otras partes del código, lo cual era más difícil que ocurriera en el anterior tutor.

Por otro lado, el trabajo ha supuesto una nueva experiencia a la hora de reconstruir y rediseñar un código ya existente, así como el aprendizaje del manejo de ontologías y un nuevo lenguaje orientado a objetos, el lenguaje C#.

Como trabajos futuros, cabría destacar que el nuevo sistema automático de tutoría formará parte de un sistema de tutoría más potente que ofrecerá una tutoría más personalizada para cada estudiante y que se encuentra en fase de diseño y desarrollo en estos momentos. El sistema presentado en este trabajo fin de grado proveerá a este sistema mayor de la parte de tutoría reactiva, es decir, la tutoría que se proporcionará de manera inmediata tras la ejecución de una acción por parte del estudiante. Asimismo, el sistema presentado proveerá una ayuda auxiliar a una extensión de este sistema de tutoría más potente que se encargará de manejar un NPC como una posible asistencia por parte del tutor. Este NPC será el encargado de mostrar a los estudiantes cómo se debe realizar una parte de una determinada práctica, si el sistema de tutoría personalizada lo cree conveniente. De esta manera, no sólo se podrán utilizar mensajes de texto para ayudar a los estudiantes, sino también demostraciones prácticas. También cabría señalar que este nuevo sistema de tutoría más potente se beneficiará de la ontología integrada dentro del sistema de tutoría desarrollado en este trabajo fin de grado, ya que gracias a los mecanismos internos e inherentes de las ontologías, este nuevo sistema podrá realizar inferencias sobre las trazas de los estudiantes para proporcionar una tutoría aún más personalizada, además de dotar al sistema de tutoría desarrollado de la capacidad de recuperación tras una caída por una causa externa.

Bibliografía

[Clemente et al., 2011] Clemente, J., Ramírez, J., and De Antonio, A. (2011) A proposal for student modeling based on ontologies and diagnosis rules. Expert Systems with Applications 38 (2011) 8066-8078.

[Fdez-Avilés, 2013] Fdez-Avilés, D. (2013) Trabajo Fin de Grado: Diseño de una práctica para un laboratorio virtual de biotecnología multilingüe y adaptable a alumnos de secundaria. Tutor: Jaime Ramírez Rodríguez, Grado de Ingeniería Informática en la Escuela Técnica Superior de Ingeniería Informática de la U.P.M.

[Linden Research, Inc., 2014a] Linden Research Inc. (2014a) What is Second Life? http://secondlife.com/whatis/ [Acceso: 04-11-2014].

[Linden Research, Inc., 2014b] Linden Research Inc. (2014b) About Linden Lab. http://www.lindenlab.com/about [Acceso: 04-11-2014].

[Linden Research, Inc., 2014c] Linden Research Inc. (2014c) Linden Scripting Language. http://wiki.secondlife.com/wiki/LSL_Portal [Acceso: 04-11-2014].

[Linden Research, Inc., 2014d] Linden Research Inc. (2014d) History. http://wiki.secondlife.com/wiki/History [Acceso: 04-11-2014].

[Linden Research, Inc., 2014e] Linden Research Inc. (2014e) Linden Dollar. http://wiki.secondlife.com/wiki/Linden_Dollar [Acceso: 04-11-2014].

[Linden Research, Inc., 2014f] Linden Research Inc. (2014f) LindenX. http://wiki.secondlife.com/wiki/LindeX [Acceso: 04-11-2014].

[Linden Research, Inc., 2014g] Linden Research Inc. (2014g) Marketplace. https://marketplace.secondlife.com/ [Acceso: 04-11-2014].

[OpenSim Team, 2014a] OpenSim Team (2014a). What is OpenSimulator? http://opensimulator.org/wiki/Main_Page [Acceso: 04-11-2014].

[OpenSim Team, 2014b] OpenSim Team (2014b). History. http://opensimulator.org/wiki/History [Acceso: 04-11-2014].

[Riofrío, 2012] Riofrío, D. (2012) Diseño e implementación de un laboratorio virtual de biotecnología. http://oa.upm.es/13700/

[Roque, 2013] Roque, A. (2013) Tesis de Máster: Implementación de una práctica virtual de biotecnología: la transformación bacteriana. Director/es: Jaime Ramírez Rodríguez, Máster de Software y Sistemas de la Facultad de Informática de la U.P.M.

[The Apache Software Foundation, 2014] The Apache Software Foundation, 2014 (2014) What is Jena? http://jena.apache.org/about_jena/about.html [Acceso: 05-11-2014].

[Unity Technologies, 2014] Unity Technologies, 2014 (2014) Unity 3D http://unity3d.com/es [Acceso: 28-12-2014].

[Wikimedia Foundation Inc. 2014a] Wikimedia Foundation Inc. (2014) Virtual World. http://en.wikipedia.org/wiki/Virtual_world [Acceso: 05-11-2014].

[Wikimedia Foundation Inc. 2014b] Wikimedia Foundation Inc. (2014) Multi-User Dungeon. http://en.wikipedia.org/wiki/MUD [Acceso: 05-11-2014].

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Wed Jan 07 22:47:30 CET 2015
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)